Binary Search Tree (BST)

A Recitation of CLRS Chapter 12

Wei Peng

IUPUI

18 September 2013

we are going to address two things today

- **understand** BST
- **how** to write (pseudo)code
  - a.k.a., how to **ace** quizzes

let us try **understanding BST** first

motivation

order

which of the following two ways is faster for understanding the word "asymptotic?" **why**?

- look up in a **bound** dictionary.
- sift through pages of a dictionary, **unbound and shuffled**.

please do not cheat by saying "I googled it"☺

which of the following two ways is faster for understanding the word "asymptotic?" **why**?

- look up in a **bound** dictionary.
- sift through pages of a dictionary, **unbound and shuffled**.

please do not cheat by saying "I googled it"☺

order helps us speed up many tasks, e.g., linear vs. binary search

order matters!

motivation

the best of two worlds

- sorted array
  - fast: search ($O(\lg n)$)
  - slow: insertion/deletion ($O(n)$)
- unsorted linked list
  - fast: insertion/deletion ($O(1)$[1])
  - slow: search ($O(n)$)

how to do fast **search** and **insertion/deletion** at the **same time**?

[1] $O(1)$ for deletion is based on the assumption that the element to be deleted has already been located. otherwise, it is dominated by the $O(n)$ search to locate it.

motivation

think: what **structural properties** make **search** and **insertion/deletion** fast?

motivation

think: what **structural properties** make **search** and **insertion/deletion** fast?

- insertion/deletion: pointers
- search: the ability to quickly find the median of smaller/larger numbers

binary search tree (BST)

BSTs embody these structural properties

- use **pointers** to organize dynamic structure
- point to (hopefully) **median** of smaller/larger numbers
  - the catch is that it is **nontrivial** to maintain the "point-to-median" property
  - vanilla BST is at the mercy of insertion/deletion order
  - self-balancing BSTs[1] (e.g., red-black tree and splay tree) try harder to come closer to the ideal

[1] `http://goo.gl/7Kap2q`

binary search tree (BST)
definition

BST is a **tree plus** the BST **order property**

- **smaller** nodes are on the **left** subtree
- **larger** nodes are on the **right** subtree

what makes BST fast

- operations descend through the tree hierarchy
- therefore, complexity is dominated by the **height** of tree
- many tree has a height of $O(\lg n)$

the **recursive** nature of trees makes it natural to process BST **recursively**

binary search tree (BST)

supported operations and algorithmic solutions

- walk ($O(n)$): properly schedule visit to **this**, **left**, and **right**
- minimum/maximum ($O(\lg n)$[1]): find leftist/rightest node
- search ($O(\lg n)$): go left/right for smaller/larger node
- predecessor/successor ($O(\lg n)$): find the maximum/minimum in left/right subtree *or* the **first** smaller/larger ancestor
- insert ($O(\lg n)$): search proper position and put in place
- delete ($O(\lg n)$): move predecessor or successor here

[1]time complexity for (almost) perfectly balanced BST; skewed BST can degenerate to O(n); same comment applies for following cases

now, let us see **how** to write (pseudo)code

technique

- strategy vs tactic
  - strategy: program organization and interface specification
  - tactic: fleshing out details
- get big picture right before messing with details
  - translate English logical structure to code template
    - identifying common patterns: conditionals, iterations, recursions
  - write code by layers
    - example: C programmers write "{" and "}" in pairs
- wishful thinking
  - treat other procedures (even unfinished) as blackboxs
  - reveal which procedures need to be written
- "premature optimization is the root of all program evils"
  - write simple/clean code first before going for performance
  - do not rush for iteration if recursion is straightforward
    - some recursions can be translated to iterations by simulating stacks
    - some language compilers can automatically optimize tail recursions to iterations

to see is to believe

- C is not a good language for learning algorithm
  - it forces you to think at a relatively low abstraction level
  - you have to fiddle with memory (de)allocation if you go beyond static data structures
  - it is more like sugar-coated assembly
  - nevertheless, it is the status quo in system programming and you have no other choice if you hack[2] Linux kernel
- **Python** is a popular alternative; **Scheme** is an excellent choice
- I am going to show you how I code BST in **Common Lisp**
- if you do not speak Common Lisp, treat it as a kind of **executable pseudo-code**
- the point is to demonstrate the **process** of coding

[2] `http://goo.gl/6pazHs`

caveat emptor

> *Thus, programs must be written for people to read, and only incidentally for machines to execute.*
>
> *Abelson, Sussman & Sussman, SICP*[3]

- the code you are about to see are optimized for readability/simplicity, not performance
- you do not have to memorize the code, it is easier to reinvent the code than to memorize it **once you understand it**
- nevertheless, follow the procedure once or twice just for the experience

[3] `http://goo.gl/04Iq1P`

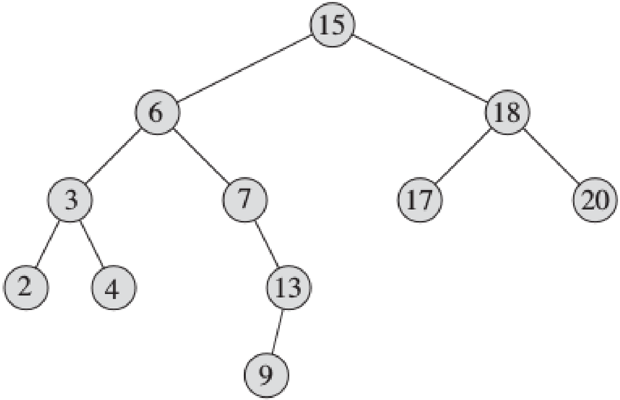
know what to do

- nodes in BST have:
  - value
  - left and right children
  - (optional) parent
- BST encodes order, so can be used for querying:
  - minimum and maximum
  - predecessor and successor
- constructing/manipulating BST require:
  - inserting a node
  - deleting a node

  that preserve BST order property
- search a node with a given value

let us get started now

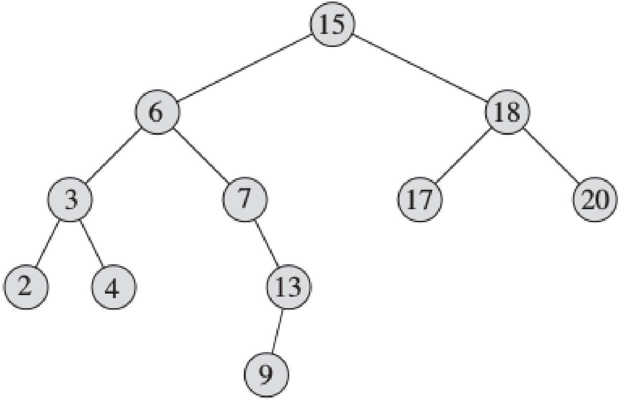
running example



CLRS3 Figure 12.2 on Page 209

Exercise

given a BST (e.g., the running example), find **an** *order of insertions that construct that tree*



▸ Solution

auxiliaries

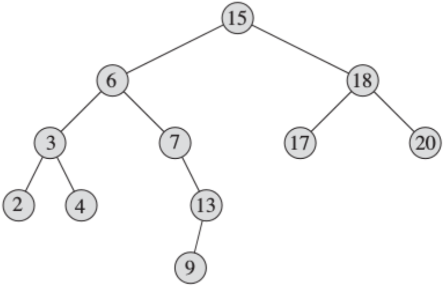
definition

```lisp
(defstruct (bst-node
            ;; for pretty printing
            (:print-object
             (lambda (tree stream)
               (labels ((bst-representation (tree) ; convert tree to a list
                          (if tree
                              (list (bst-node-value tree) ; the value
                                    ;; descend into the children
                                    (bst-representation (bst-node-left-child tree))
                                    (bst-representation (bst-node-right-child tree)))
                              nil)))
                 (pprint-logical-block (stream nil
                                               :prefix #.(format nil "#<BST-NODE~%")
                                               :suffix #. (format nil "~%>"))
                   (labels ((bst-pprint (tree)
                              (pprint-logical-block (stream nil)
                                (let ((value (first tree)))
                                  (when value
                                    (write value :stream stream)
                                    (pprint-tab :section-relative 1 3 stream)
                                    (pprint-logical-block (stream (rest tree)) ; the children
                                      (let ((left-child (pprint-pop))
                                            (right-child (pprint-pop)))
                                        (when (or left-child right-child)
                                          (if left-child
                                              (bst-pprint left-child)
                                              (princ "*" stream))
                                          (when right-child
                                            (pprint-newline :mandatory stream)
                                            (bst-pprint right-child))))))))))
                     (bst-pprint (bst-representation tree))))))))
  ;; real meat is here
  value
  parent
  left-child
  right-child)
```

the hairy stuff in the middle is for "pretty printing" the structure

auxiliaries

pretty printing




```
BINARY-SEARCH-TREE> *bst-example*
#<BST-NODE
15 6  3  2
         4
      7  *
         13 9
   18 17
      20
>
```

auxiliaries

build BST from values

```lisp
(defun bst-insert-nodes-from-values (tree &rest values)
  (loop for value in values do
    (setf tree (bst-insert-node tree (make-bst-node :value value)))
    (when *bst-show-details* (write tree :pretty t) (terpri)))
  tree)

(defun bst-delete-nodes-from-values (tree &rest values)
  (loop for value in values do
    (setf tree (bst-delete-node tree (bst-search-node tree value)))
    (when *bst-show-details* (write tree :pretty t) (terpri)))
  tree)

(defun build-bst-from-values (&rest values)
  (let ((tree nil))
    (apply #'bst-insert-nodes-from-values tree values)))

(defparameter *bst-example*  nil
  "CLRS3 Figure 12.2 on Page 290")

(defvar *bst-root* nil
  "root of the BST")

(defun init ()
  (setf *bst-example*
        (build-bst-from-values
         15 6 18 3 7 17 20 2 4 13 9))
  (setf *bst-root* nil))

(init)
```

preorder, inorder, and postorder walk

```lisp
(defun bst-inorder-map (tree &optional (f #'identity))
  (if tree
      (nconc (bst-inorder-map (bst-node-left-child tree) f)
             (list (funcall f (bst-node-value tree)))
             (bst-inorder-map (bst-node-right-child tree) f))
      nil))

(defun bst-preorder-map (tree &optional (f #'identity))
  (if tree
      (nconc (list (funcall f (bst-node-value tree)))
             (bst-preorder-map (bst-node-left-child tree) f)
             (bst-preorder-map (bst-node-right-child tree) f))))

(defun bst-postorder-map (tree &optional (f #'identity))
  (if tree
      (nconc (bst-postorder-map (bst-node-left-child tree) f)
             (bst-postorder-map (bst-node-right-child tree) f)
             (list (funcall f (bst-node-value tree))))))
```

Exercise

Exercise

which one among preorder, inorder, and postorder walk is equivalent to sorting?

▸ Solution

minimum/maximum and predecessor/successor

```lisp
(defun bst-minimum (tree)
  "find leftist node"
  (let ((left-child (bst-node-left-child tree)))
    (if left-child
        (bst-minimum left-child)
        tree)))

(defun bst-maximum (tree)
  "find rightest node"
  (let ((right-child (bst-node-right-child tree)))
    (if right-child
        (bst-maximum right-child)
        tree)))

(defun bst-predecessor (tree)
  "find the bst-maximum in left subtree or first smaller ancestor"
  (let ((left-child (bst-node-left-child tree)))
    (if left-child
        (bst-maximum left-child)
        (do ((node tree parent)
             (parent (bst-node-parent tree) (bst-node-parent parent)))
            ((or (null parent)
                 (eql (bst-node-right-child parent) node))
             node)))))

(defun bst-successor (tree)
  "find the bst-minimum in right subtree or first larger ancestor"
  (let ((right-child (bst-node-right-child tree)))
    (if right-child
        (bst-minimum right-child)
        (do ((node tree parent)
             (parent (bst-node-parent tree) (bst-node-parent parent)))
            ((or (null parent)
                 (eql (bst-node-left-child parent) node))
```

minimum/maximum and predecessor/successor

```
BINARY-SEARCH-TREE> (bst-minimum *bst-example*)
#<BST-NODE
2  >
BINARY-SEARCH-TREE> (bst-maximum *bst-example*)
#<BST-NODE
20 >
BINARY-SEARCH-TREE> (bst-predecessor *bst-example*)
#<BST-NODE
13 9  >
BINARY-SEARCH-TREE> (bst-successor *bst-example*)
#<BST-NODE
17 >
BINARY-SEARCH-TREE>
```

search a node

```
(defun bst-search-node (tree value &key (key #'identity) (test-eql #'equal) (test-ord #'<))
  (if tree
      (let* (
             (node-value (bst-node-value tree))
             (node-value-key (funcall key node-value))
             (value-key (funcall key value))
             )
        ;(format *trace-output* "~&~A ~A ~A" node-value node-value-key value-key)
        (if (funcall test-eql node-value-key value-key)
            tree
            (if (funcall test-ord value-key node-value-key)
                (bst-search-node (bst-node-left-child tree) value
                             :key key :test-eql test-eql :test-ord test-ord)
                (bst-search-node (bst-node-right-child tree) value
                             :key key :test-eql test-eql :test-ord test-ord))))
      nil))
```

search a node

```
BINARY-SEARCH-TREE> *bst-example*
#<BST-NODE
15 6  3  2
         4
      7  *
         13 9
   18 17
      20
>
BINARY-SEARCH-TREE> (bst-search-node *bst-example* 7)
#<BST-NODE
7  *
   13 9
>
BINARY-SEARCH-TREE>
```

insert a node

```lisp
(defun bst-insert-node (tree node &key (test-ord #'<))
  (labels ((bst-insert-node! (tree node &key parent leftp (test-ord #'<))
             "*BST-ROOT* should be bound to the root before calling BST-INSERT-NODE!"
             (if tree
                 (let ((leftp (funcall test-ord
                                       (bst-node-value node)
                                       (bst-node-value tree))))
                   (bst-insert-node! (if leftp
                                         (bst-node-left-child tree)
                                         (bst-node-right-child tree))
                                     node
                                     :parent tree
                                     :leftp leftp
                                     :test-ord test-ord)
                   tree)
                 (progn
                   (setf (bst-node-parent node) parent)
                   (if parent
                       (if leftp
                           (setf (bst-node-left-child parent) node)
                           (setf (bst-node-right-child parent) node)))
                   node))))
    (let ((*bst-root* tree))
      (setf *bst-root*
            (bst-insert-node! *bst-root* node :test-ord test-ord))
      *bst-root*)))
```

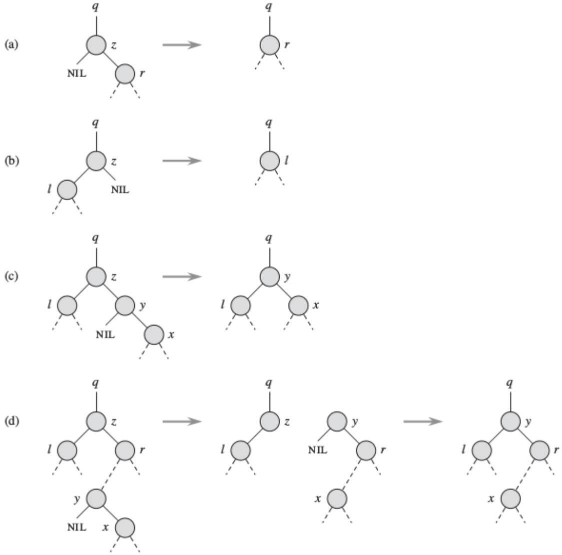
insert a node

Q: what is the process of building a tree from the sequence 5, 3, 2, 6, 9, 8, and 7?

insert a node

```
BINARY-SEARCH-TREE> (let ((*BST-SHOW-DETAILS* t))
                      (build-bst-from-values 5 3 2 6 9 8 7))
#<BST-NODE
5   >
#<BST-NODE
5   3   >
#<BST-NODE
5   3   2   >
#<BST-NODE
5   3   2
    6
>
#<BST-NODE
5   3   2
    6   *
        9
>
#<BST-NODE
5   3   2
    6   *
        9   8
>
#<BST-NODE
5   3   2
    6   *
        9   8   7
>
#<BST-NODE
5   3   2
```

delete a node



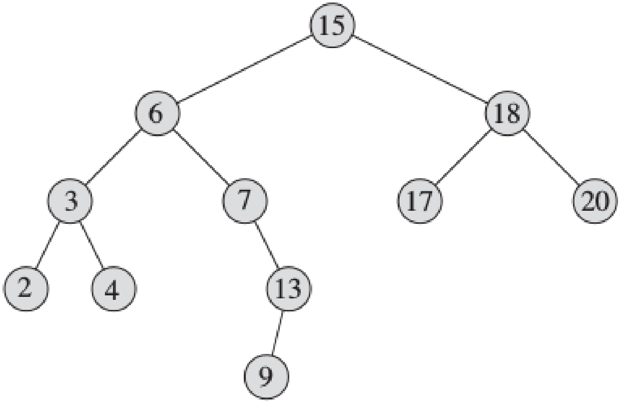
CLRS3 Figure 12.4 on Page 297

delete a node

```
(defun bst-delete-node (tree node)
  (labels ((bst-delete-node! (node)
             "*BST-ROOT* should be bound to the root before calling BST-DELETE-NODE!"
             (flet ((transplant (node other-node)
                      (let ((node-parent (bst-node-parent node)))
                        (if node-parent
                            (progn
                              (if (eql node (bst-node-left-child node-parent))
                                  (setf (bst-node-left-child node-parent) other-node)
                                  (setf (bst-node-right-child node-parent) other-node))
                              (when other-node
                                (setf (bst-node-parent other-node) node-parent))
                              node-parent)
                            (progn                          ; when node is the root
                              (when other-node
                                (setf (bst-node-parent other-node) nil))
                              (setf *bst-root* other-node))))))
               (if (bst-node-left-child node)
                   (if (bst-node-right-child node)
                       (let ((node-successor (bst-successor node)))      ; the hard case
                         (unless (eql node (bst-node-parent node-successor)) ; the hardest case
                           (transplant node-successor (bst-node-right-child node-successor))
                           (let ((node-right-child (bst-node-right-child node)))
                             (setf (bst-node-right-child node-successor) node-right-child)
                             (setf (bst-node-parent node-right-child) node-successor)))
                         (transplant node node-successor)
                         (let ((node-left-child (bst-node-left-child node)))
                           (setf (bst-node-left-child node-successor) node-left-child)
                           (setf (bst-node-parent node-left-child) node-successor)))
                       (transplant node (bst-node-left-child node))) ; the easy case: right child nil
                   (transplant node (bst-node-right-child node)) ; the easy case: left child nil
                   ))))
    (let ((*bst-root* tree))
      (bst-delete-node! node)
      *bst-root*)))
```

delete a node

Q: What is the process of deleting 15, 18, and 13 in order from the running example



delete a node

```
BINARY-SEARCH-TREE> *bst-example*
#<BST-NODE
15 6  3  2
         4
      7  *
         13 9
   18 17
      20
>
BINARY-SEARCH-TREE> (let ((*BST-SHOW-DETAILS* t))
                      (bst-delete-nodes-from-values *bst-example* 15 18 13))
#<BST-NODE
17 6  3  2
         4
      7  *
         13 9
   18 *
      20
>
#<BST-NODE
17 6  3  2
         4
      7  *
         13 9
   20
>
#<BST-NODE
17 6  3  2
         4
      7  *
         9
   20
>
#<BST-NODE
17 6  3  2
         4
      7  *
```

Q & A

to try the code, follow the instruction

`https://github.com/pw4ever/cl-algo-n-struct` reference solution

Solution to Exercise

Proof.
(hint) later-inserted elements go **down** along the tree hierarchy.
therefore, parents are inserted earlier than children.
“an” rather than **“the”** order: sibling insertion order can be arbitrary
a solution: pre-order walk $\square$

Solution to ▸ Exercise

Proof.
let us see.

```
BINARY-SEARCH-TREE> (bst-preorder-map *bst-example*)
(15 6 3 2 4 7 13 9 18 17 20)
BINARY-SEARCH-TREE> (bst-inorder-map *bst-example*)
(2 3 4 6 7 9 13 15 17 18 20)
BINARY-SEARCH-TREE> (bst-postorder-map *bst-example*)
(2 4 3 9 13 7 6 17 20 18 15)
BINARY-SEARCH-TREE>
```

□