CSE 101 5-4-21

Runtime of BFS: (worst case)

basic operations: assignments
comparisons
queue operations

size of input graph:

n = # vertices
m = # edges

Cost:

initialize: $\Theta(n)$

queue ops: $\Theta(n)$

scanning adj lists: Θ(tot. len. of adj lists)
$= \Theta$(sum of vert. degrees)
$= \Theta(m)$

Total cost of BFS:

$= \Theta(n) + \Theta(n) + \Theta(m)$

$= \Theta(n+m)$

note: # of bytes needed to store adj list rep. is

$$\alpha n + \beta m = \Theta(n+m)$$

BFS is linear in size of adj list rep.

Runtime of DFS:

basic ops: assignments
comparisons.

size of input: n, m

cost:

initialize: $\Theta(n)$

main loop: $\Theta(n)$ (apart from visit(x))

calls to visit: $\Theta(n)$

scanning adj lists:

Θ(total len. of all adj lists)

$= \Theta$(sum of deg-cer)

$= \Theta(m)$

total cost: $\Theta(n)+\Theta(n)+\Theta(n)+\Theta(m)$

$= \Theta(n+m)$ (linear in # bytes storing adj lists)

Binary Search Tree: (BST)

Chap 12 in CLRS.

Dictionary ADT:

state: a set of ordered pairs:

(key, value)

i.e. a ~~state~~ is

$\{(K_1, V_1), (K_2, V_2), \ldots, (K_n, V_n)\}$

option: keys are distinct.

operations:

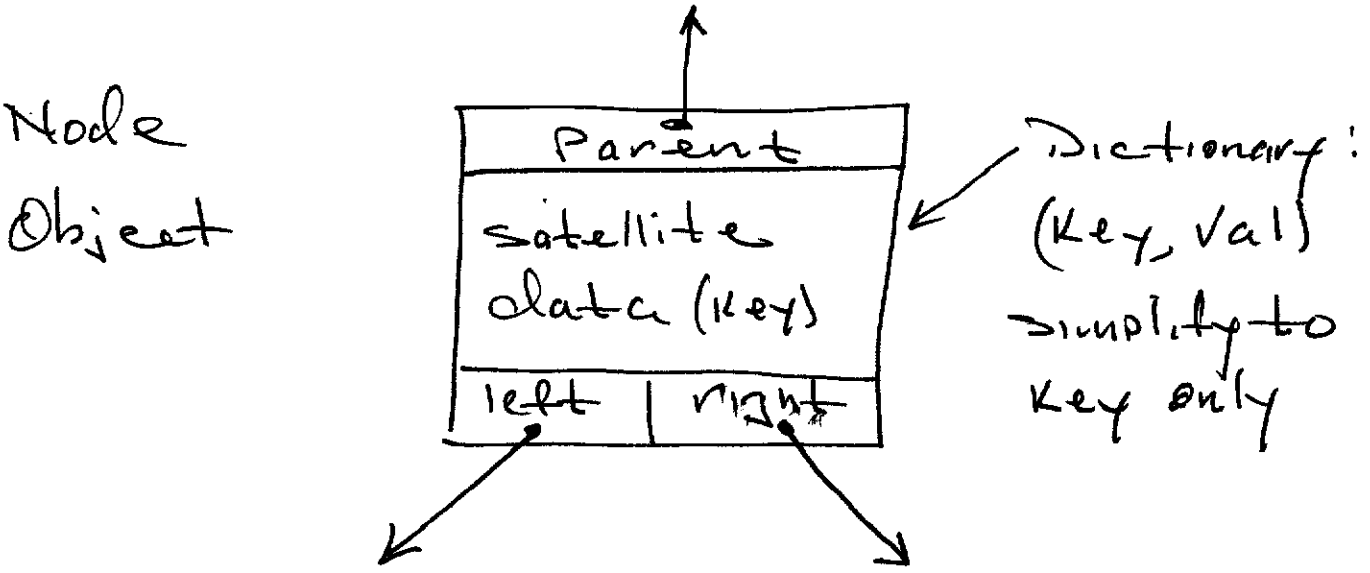
- lookup(key): return value assoc with key if (key, val) is in dictionary. return nil otherwise.

- insert(key, val): add a new pair (key, val) to dictionary. [if distinct keys: precondition lookup(key) == nil .]
- delete(key): remove (key, val) from dictionary. [precondition: lookup(key) ≠ nil .]

one way to implement Dictionary:

Binary Search Tree (BST)

A BST is a linked data structure



root: only node with nil parent

leaf: any node with no children

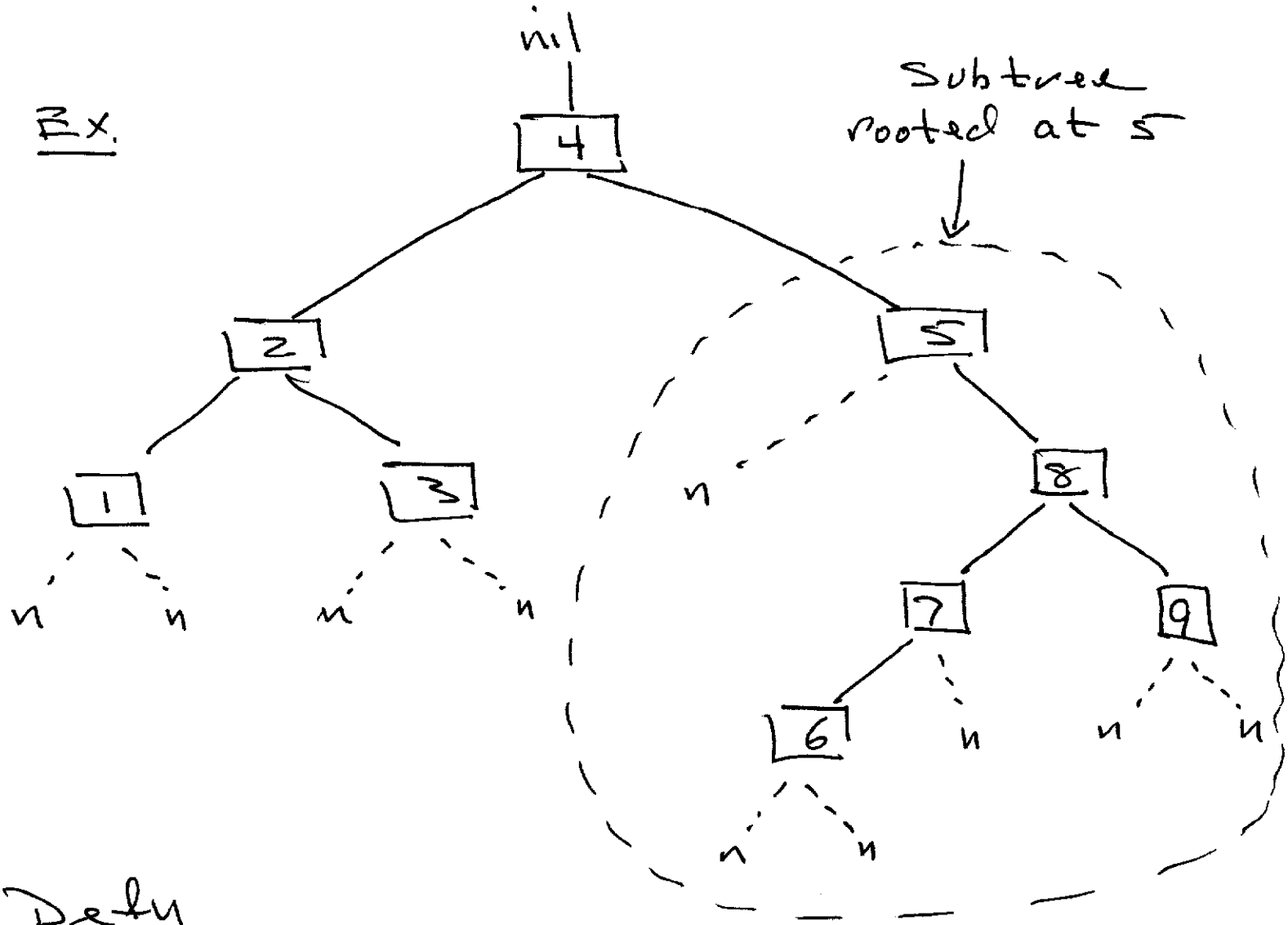
BST Properties

Let x, y be nodes in a BST. Then

(1) if y is in the left sub-tree of x (i.e. y is a descendant of left[x]) then $key[y] \leq key[x]$.

(2) if y is in right subtree of x (y is a desc. of right[x]), then $key[x] \leq key[y]$.

Ex.



Defn

- a leaf is a node with no children
- an internal node a non-leaf.

Implementation note:

recommend that we have a dummy node called nil as a field in DictionaryObj.

Tree traversals:

- InOrderTreeWalk(x)
- PreOrderTreeWalk(x)
- PostOrderTreeWalk(x)

all recursive.

all Print keys in subtree rooted at x.

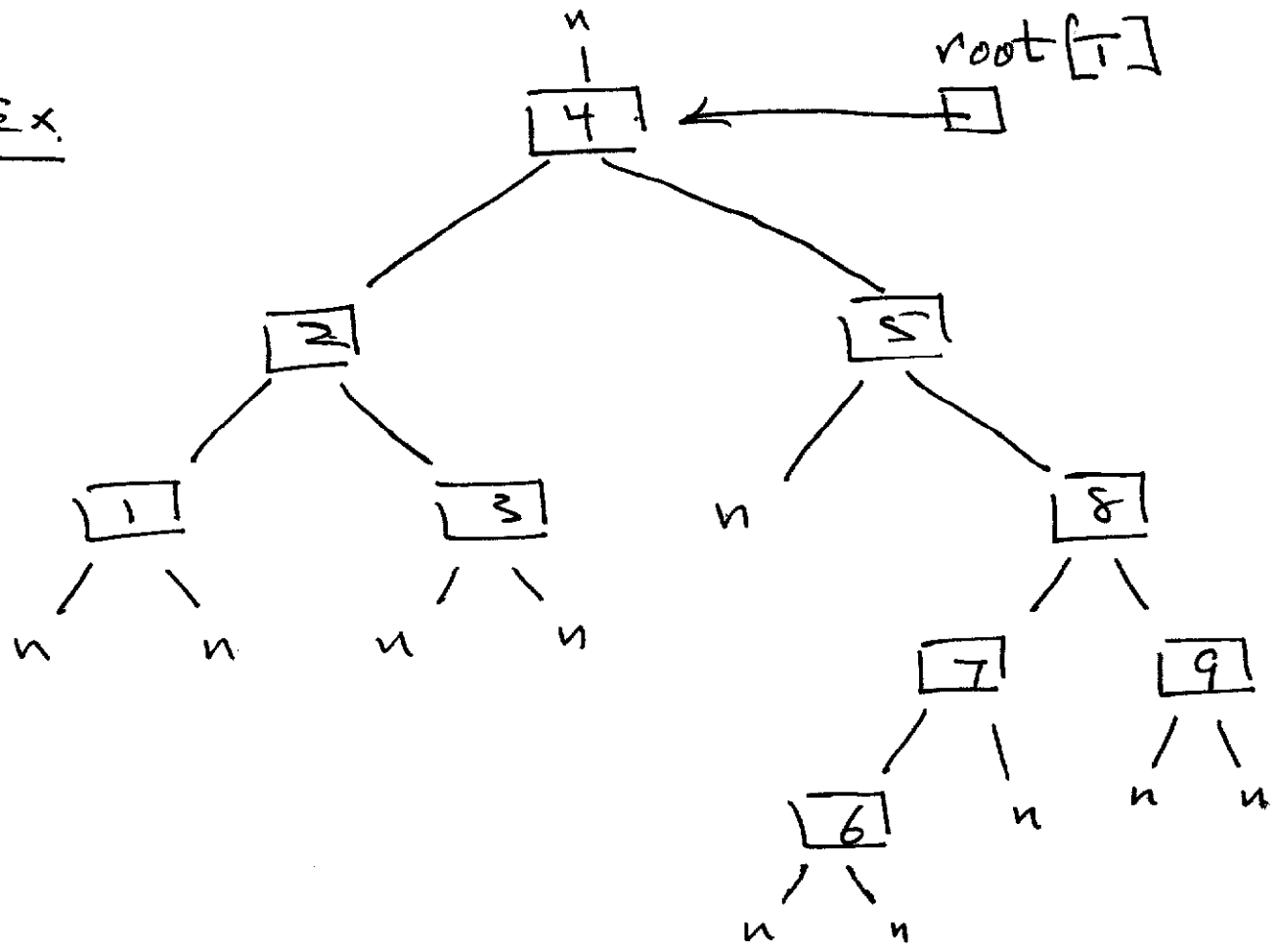
InOrderTreeWalk(x)

1. if $x \neq nil$
2. InOrderTreeWalk(left[x])
3. print(key[x])
4. InOrderTreeWalk(right[x])

book: x.right
c: x -> right
c++: x -> right

any kind of processing of satellite data at x can occur here

Ex.



InOrderTreeWalk(root[T]):

output: 1 2 3 4 5 6 7 8 9

Runtime: Θ(# of nodes)

$= \Theta(n)$.

PreOrderTreeWalk(x)

1. if $x \neq nil$.
2. print(key[x])
3. PreOrderTreeWalk(left[x])
4. PreOrderTreeWalk(right[x])

Ex. 4 2 1 3 5 8 7 6 9

Runtime: $\Theta(n)$ (n = #nodes)

Exercise: write PostOrderTreeWalk(x)

Ex. 1 3 2 6 7 9 8 5 4

Part : assignment =