12.1 BINARY SEARCH TREES

A BINARY SEARCH TREE (BST) IS A BT FOR WHICH EACH NODE x CONTAINS FIELDS:

$$key[x],\ left[x],\ right[x],\ p[x]$$ (+ SATELLITE DATA)

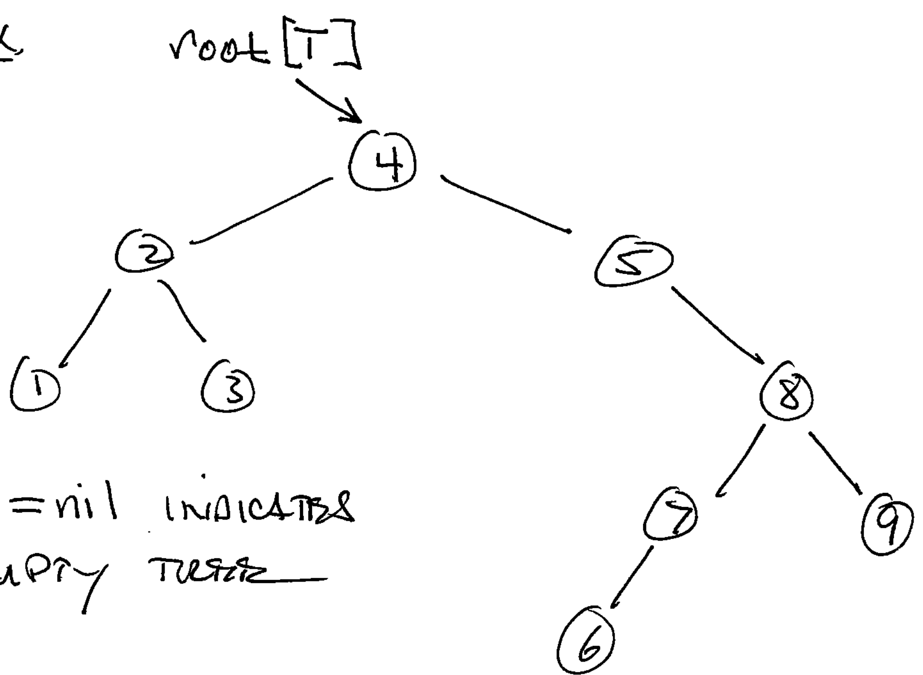
AND SATISFYING THE

BINARY SEARCH TREE PROPERTIES:

FOR ALL NODES x:

- IF y IS IN THE LEFT SUBTREE OF x, THEN $key[y] \le key[x]$.
- IF y IS IN THE RIGHT SUBTREE OF x, THEN $key[x] \le key[y]$.

EX



THE BST PROPERTY MAKES IT POSSIBLE TO PRINT OUT KEYS IN ASCENDING ORDER

InOrderTreeWalk(x)

1.) if $x \neq nil$
2.) InOrderTreeWalk(left[x])
3.) Print key[x] (COULD DO OTHER PROCESSING OF X HERE)
4.) InOrderTreeWalk(right[x])

TO PRINT FULL TREE, call:

InOrderTreeWalk(root[T]).

RUN TIME: $\Theta(n)$ IF T HAS n NODES. SEE PROOF ON P. 255.

A PRE-ORDER TREE WALK WOULD PRINT ~~[illegible]~~ key[x] BEFORE IT PRINTS THE VALUES IN EITHER SUBTREE.

A POST-ORDER TREE WALK PRINTS ~~THE~~ key[x] AFTER IT PRINTS VALUES IN left & right SUBTREES.

EXERCISE: WRITE PSEUDO-CODE FOR THESE OPS.

12.2 QUERIES

A BST CAN SERVE AS AN EFFICIENT STRUCTURE FOR STORING DATA. (i.e. A DATABASE).

TreeSearch(x, k)

```
1.) if  x = nil  or  k = key[x]
2.)       return x
3.) if  k < key[x]
4.)       return TreeSearch(left[x], k)
5.) else
6.)       return TreeSearch(right[x], k)
```

TO SEARCH FOR A NODE x WITH KEY k WE CALL

$$\text{TreeSearch}(root[T], k).$$

IF SUCH A NODE EXISTS (A POINTER TO) THAT NODE IS RETURNED, OTHERWISE nil IS RETURNED.

NOTE: IN MANY APPLICATIONS (e.g. DATABASES) KEYS WILL BE DISTINCT. IF T CONTAINS MULTIPLE KEYS, SOME NODE WITH MATCHING KEY IS RETURNED.

THE NODES ENCOUNTERED IN TreeSearch FORM A DOWNWARD PATH STARTING AT THE ROOT. ∴ RUN TIME IS $\Theta(h)$ (WORST CASE) WHERE $h = \text{height}(T)$.

SEE ITERATIVE VERSION ON P. 257.

TreeMinimum(x)

1.) while left[x] ≠ nil
2.) x ← left[x]
3.) return x

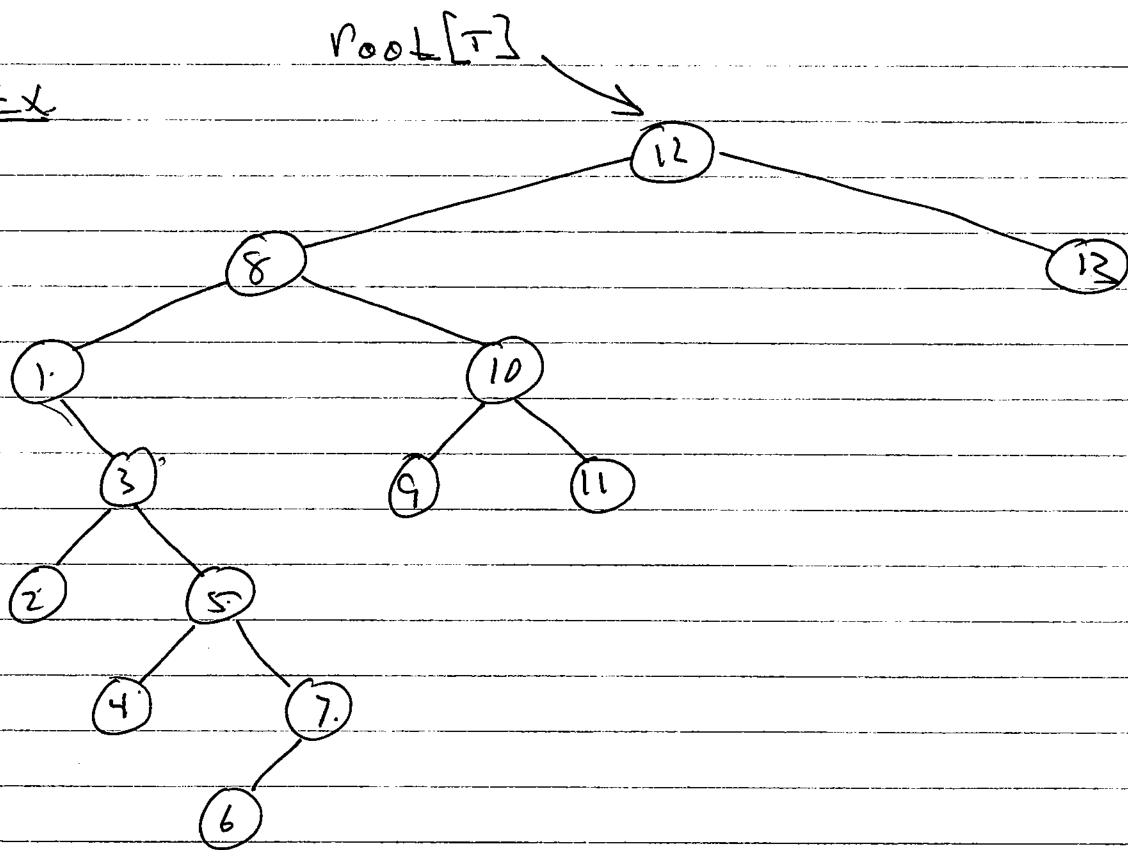
TreeMaximum(x)

1.) while right[x] ≠ nil
2.) x ← right[x]
3.) return x

CORRECTNESS OF BOTH ALGORITHMS FOLLOWS FROM BST PROPERTIES. WORST CASE RUN TIME: $\Theta(h)$ WHERE $h = \text{height}(T)$.

THE SUCCESSOR OF A NODE x IS THE NEXT NODE TO BE PROCESSED AFTER x IN AN IN ORDER TREE WALK.

Ex



SUCCESSOR(1) = 2

SUCCESSOR(8) = 9

SUCCESSOR(7) = 8

PROB 12.2-6

IF x HAS NO RIGHT CHILD & x HAS A SUCCESSOR y, THEN y IS THE LOWEST ANCESTOR OF x WHOSE LEFT CHILD IS ALSO AN ANCESTOR OF x.

TreeSuccessor(x)

```
1.) if right[x] ≠ nil
2.)       return TreeMinimum(right[x])
3.) y ← P[x]
4.) while y ≠ nil & x = right[y]
5.)       x ← y
6.)       y ← P[y]
7.) return y
```

TO UNDERSTAND THE OPERATION OF THIS ALGORITHM, WE CONSIDER TWO CASES.

CASE 1: IF RIGHT SUBTREE AT x IS NOT EMPTY, THEN SUCCESSOR OF x IS THE LEFTMOST NODE IN x'S RIGHT SUBTREE, WHICH IS RETURNED ON LINE 2.

CASE 2: IF right SUBTREE OF x IS EMPTY, AND x HAS A SUCCESSOR, y, THEN y IS THE LOWEST ANCESTOR OF x WHOSE LEFT CHILD IS ALSO AN ANCESTOR OF x (OR x ITSELF). (SEE PROB 12.2-6) TO FIND THIS NODE WE CLIMB UP THE TREE FROM x UNTIL WE FIND A NODE WHICH IS THE LEFT CHILD OF ITS PARENT.

12.3

~~BST~~ Insertion & Deletion

The key to maintaining the BST property is to insert & delete elements carefully.

To insert a node z with $key[z] = k$ into a BST T, first set

$$\begin{cases} key[z] \leftarrow k \\ p[z] \leftarrow left[z] \leftarrow right[z] \leftarrow NIL \end{cases}.$$

Then

<u>Insert (T, z)</u>

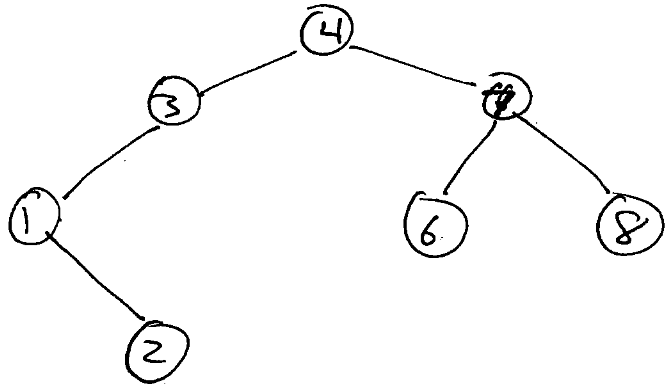
```
1.)  y ← NIL
2.)  x ← root[T]
3.)  while  x ≠ NIL
4.)          y ← x
5.)          If  key[z] < key[x]
6.)               x ← left[x]
7.)          Else
8.)               x ← right[x]
9.)  p[z] ← y
10.) If   y = NIL        (T was empty in this case)
11.)         root[T] ← z
12.) Else IF  key[z] < key[y]
13.)         left[y] ← z
14.) Else
15.)         right[y] ← z
```

THE POINTER VARIABLE x TRACES A PATH FROM THE ROOT DOWN TO A LEAF, WITH y MAINTAINED AS THE PARENT OF x. WHEN x IS SET TO NIL (in line 6 or 8), THIS NIL OCCUPIES THE POSITION WHERE z BELONGS, SO $p[z]$ is SET TO y.

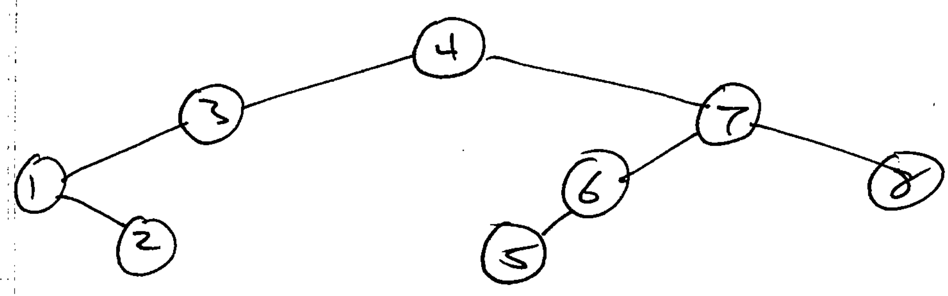
IF y IS STILL NIL AT THIS POINT THEN THE TREE WAS EMPTY TO BEGIN WITH (i.e. root[T] = NIL) SO z BECOMES THE ROOT. IF NOT, THE ONLY QUESTION IS WHETHER z SHOULD BE THE LEFT OR RIGHT CHILD OF y. THIS IS DECIDED IN LINES 12-15.

Insert TRACES A PATH FROM THE ROOT DOWN TO A LEAF, SO RUNS IN $O(h)$ TIME.

<u>EX</u>

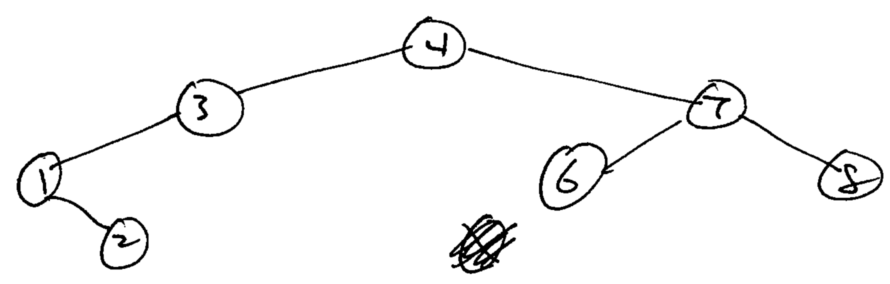


Insert(T, z) where key[z] = 5

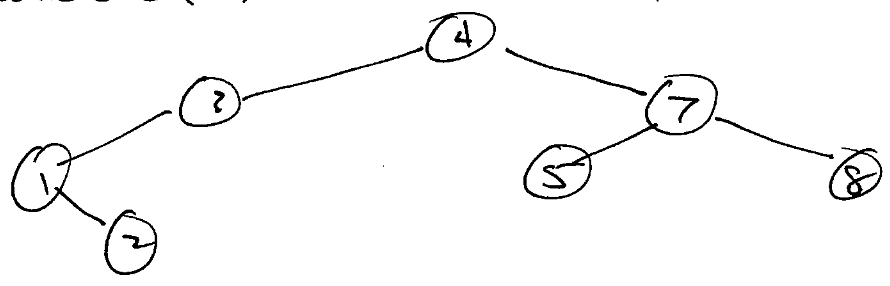


THE PROCEDURE FOR DELETING A NODE z CONSIDERS THREE CASES:

<u>CASE 1</u>: IF z HAS NO CHILDREN, MODIFY p[z] TO HAVE NIL IN PLACE OF ITS CHILD z. IN ABOVE EX. Delete(5)

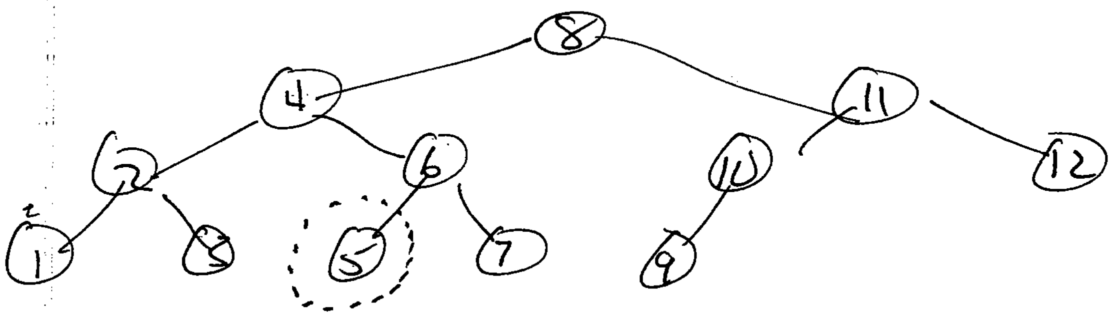


<u>CASE 2</u>: IF z HAS A SINGLE CHILD, SPLICE OUT z LINKING z's CHILD TO z's PARENT
Delete(6)

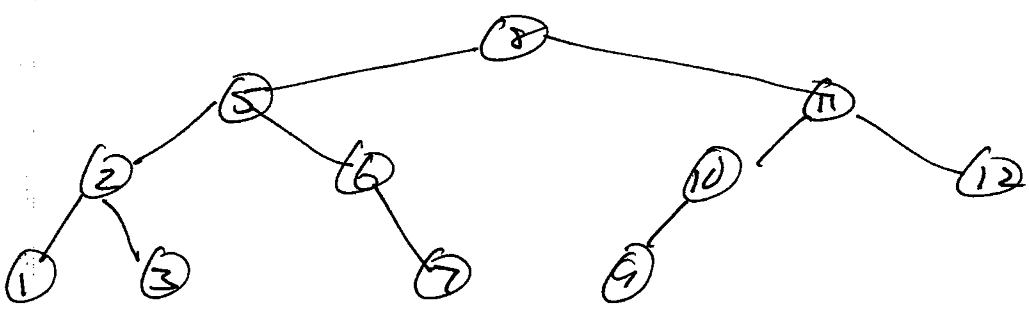


<u>CASE 3</u>: IF Z HAS TWO CHILDREN, SPLICE OUT Z'S SUCCESSOR Y (WHICH HAS NO LEFT CHILD BY PROBLEM 12.2-5) AND COPY FIELDS OF Y INTO FIELDS OF Z (i.e. KEY & SATELLITE DATA.)

<u>Ex.</u>



Delete (4). COPY DATA IN 5 INTO 4.



SEE PSEUDO-CODE P. 262

RUN TIME IS $\Theta(h)$ SINCE THAT IS TIME OF SUCCESSOR.

<u>PROBLEMS</u>

~~13.3-1 (P.253)~~
~~13.3-4 (P.254)~~
~~13.3-6 (P.254)~~

Tree-Delete P. 262

<u>NOTE</u>: EVERYTHING RUNS IN $\Theta(h)$ TIME.