Searching and Binary Search Tree (Outcome (2))

Review: let's revisit the searching problem.

Problem: Given a set of n elements and an element x, return the location of the element x or report that such an element does not exist in the set.

- Sequential (Linear) Search
- Binary Search
- Hashing

Do you remember/ learn all these

Sequential (Linear) Search

Unsorted/ sorted array implementation

Successful or unsuccessful search

- Worst case: $O(n)$
- Average case: $O(n)$

```
for i = 0 to n-1 {  //elements stored in array A[0..n-1]
    if A[i] = x
        return i;}
return -1;   // -1: not found
```

How about linked-list?

Binary Search

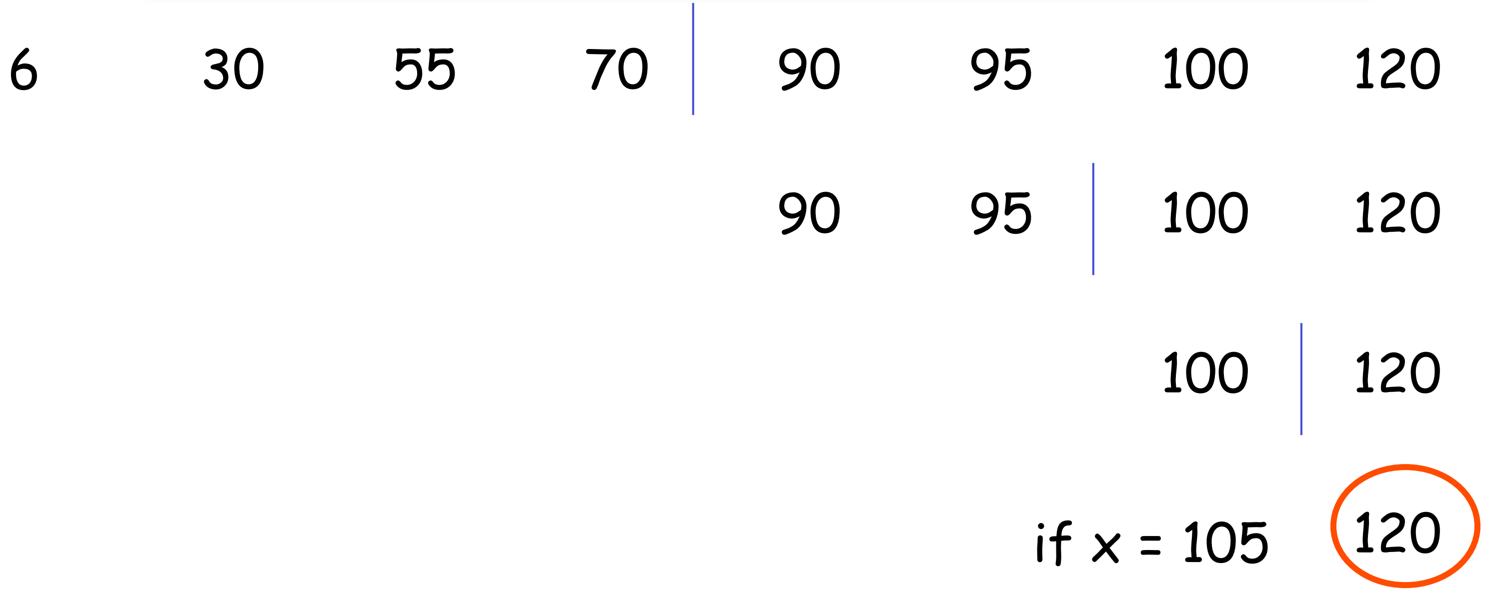
Searching in a sorted array (return i if $A[i]=x$ or -1 otherwise):

6 30 55 70 90 95 100 120 $x = 100$

Binary search:

Observation: if $(A[i] > x)$, $x \neq A[j]$ with $j>i$
if $(A[i] < x)$, $x \neq A[j]$ with $j<i$

So, check the "middle" element and eliminate half of elements if x is not found yet



```
int Search(A, x) { // binary search, A[0..n-1]
    int lo=0, hi=n-1, mid
    while (lo $\le$ hi) do
        mid = $\lfloor (lo+hi)/2 \rfloor$
        if (x < A[mid])
            hi=mid-1
        else if (x > A[mid])
            lo=mid+1
        else return mid
    return –1
}
```

(1) What is the worst case time complexity?
(2) Can you rewrite it as a recursive algorithm?

(1) The while loop will be executed O(log n) times [why?], so the time complexity of the algorithm is O(log n).

For binary search

- assuming that the elements are sorted according to their keys

Sorted array implementation

Successful or unsuccessful search

Worst case: O(log n)

Average case: O(log n) [proof?]

Sorted linked-list implementation?

Cannot implement binary search

Hashing

For ave. case complexity	Unsuccessful	Successful
Chaining	$O(1+\alpha)$	$O(1+\alpha)$
Open Addressing	$O(1/(1-\alpha))$*	$O(1/\alpha \ln(1/(1-\alpha)))$*

Worst case complexity: O(n) where n is the no. of keys in dictionary

Summary: Best worst-case algorithm – binary search
But insertion/deletion takes O(n) time, can we do better?

* Analysis not required in the exam.

Binary Search Tree

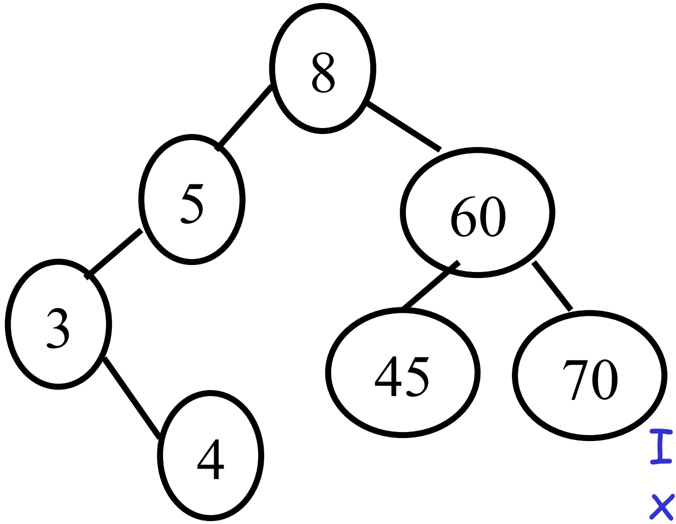
Given a set of elements, a binary search tree can support the following operations:

Search, Minimum, Maximum, Predecessor, Successor, Insert, Delete

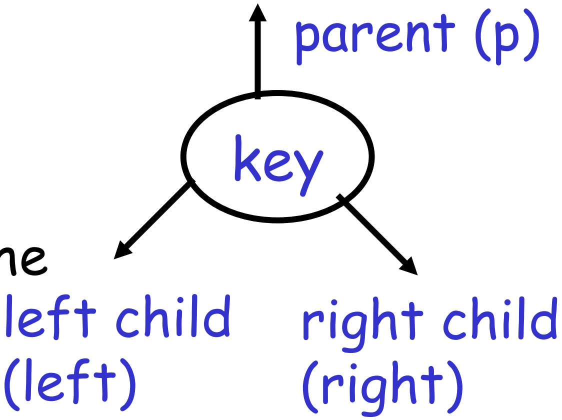
Binary search tree can be used as a dictionary!

What is a binary search tree?

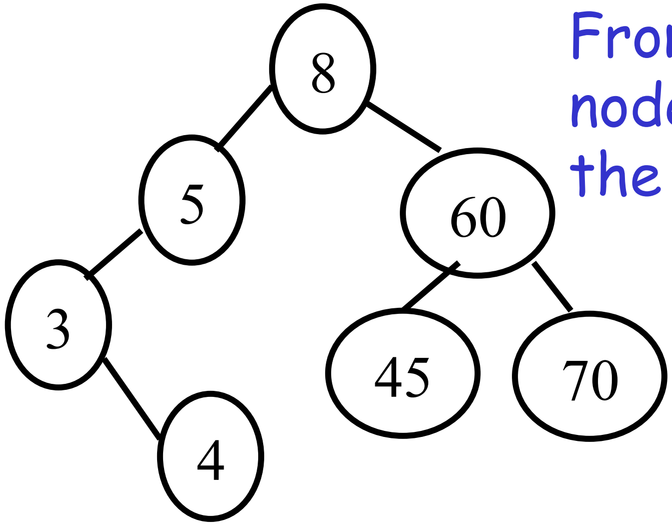
A binary search tree is a binary tree with keys stored in the nodes which satisfy the following property (binary-search-tree-property). If y and z are nodes in left subtree and right subtree, respectively, of node x, then key(x) ≥ key(y) and key(x) ≤ key(z)



We assume that each node has 3 pointers and a key stored in it. And a pointer to the root is given.



If x denotes the pointer to the node, x.p (parent), x.left (left child), x.right (right child)



From this example, can you guess which node has the minimum key? Which one has the maximum key?

Minimum: the leftmost node
Maximum: the rightmost node

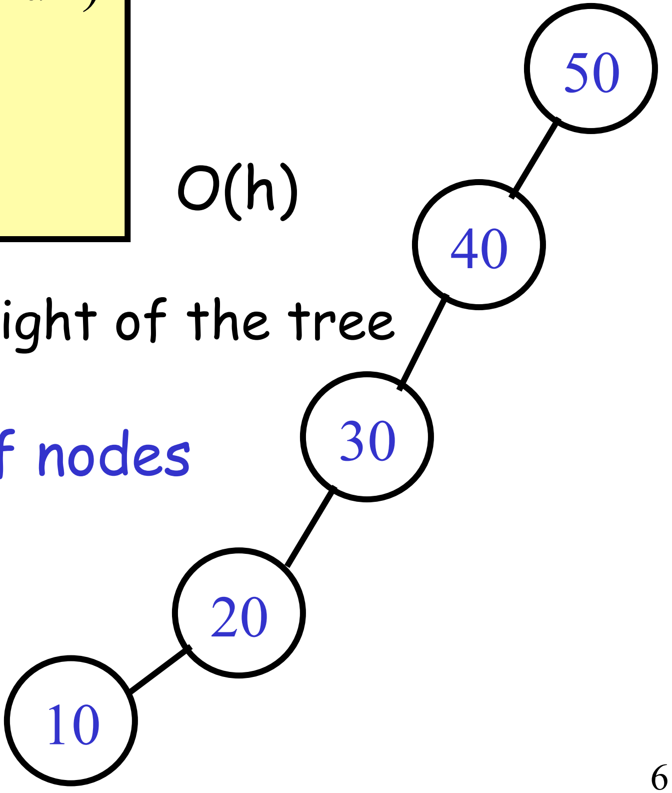
```
Tree-Maximum(x) {
   while (x.right ≠ null)
         x = x.right;
   return x;
}
```

O(h)

```
Tree-Minimum(x) {
   while (x.left ≠ null)
         x = x.left;
   return x;
}
```

O(h) where h is the height of the tree

This can be O(n) where n is the number of nodes



A worst case example
for Tree-minimum(x)

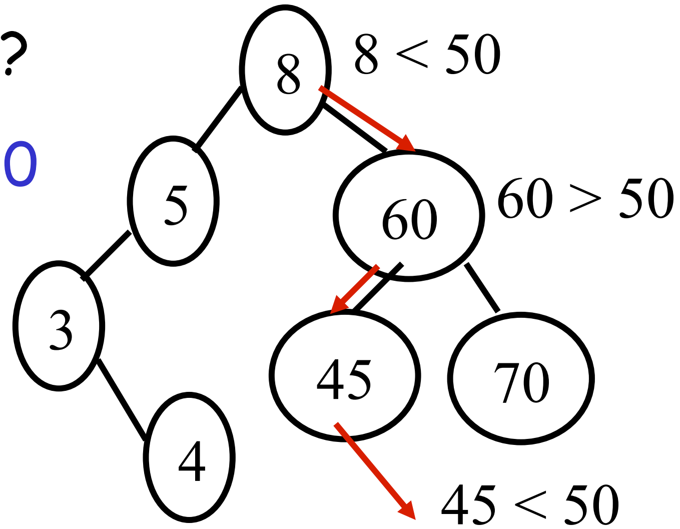
....

On the other hand, if the tree is roughly a complete binary tree, then h = O(log n)

You will see examples of such balanced search tree

Do you know how to perform search?

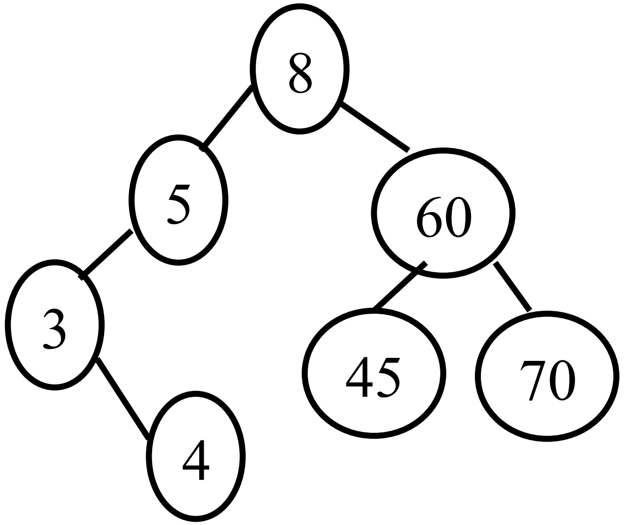
Given k = 50



```
Tree-Search(x, k) {
    if (x = Null) or (k = key(x))
        return x
    else
      if (k < key(x))
            Tree-Search(x.left, k);
      else
            Tree-Search(x.right, k);
}
```

O(h)

Do you know how to write this without using recursion?



How about Predecessor and Successor?

Successor(x): return the node whose key is just larger than that of x or Null if key(x) is the largest key

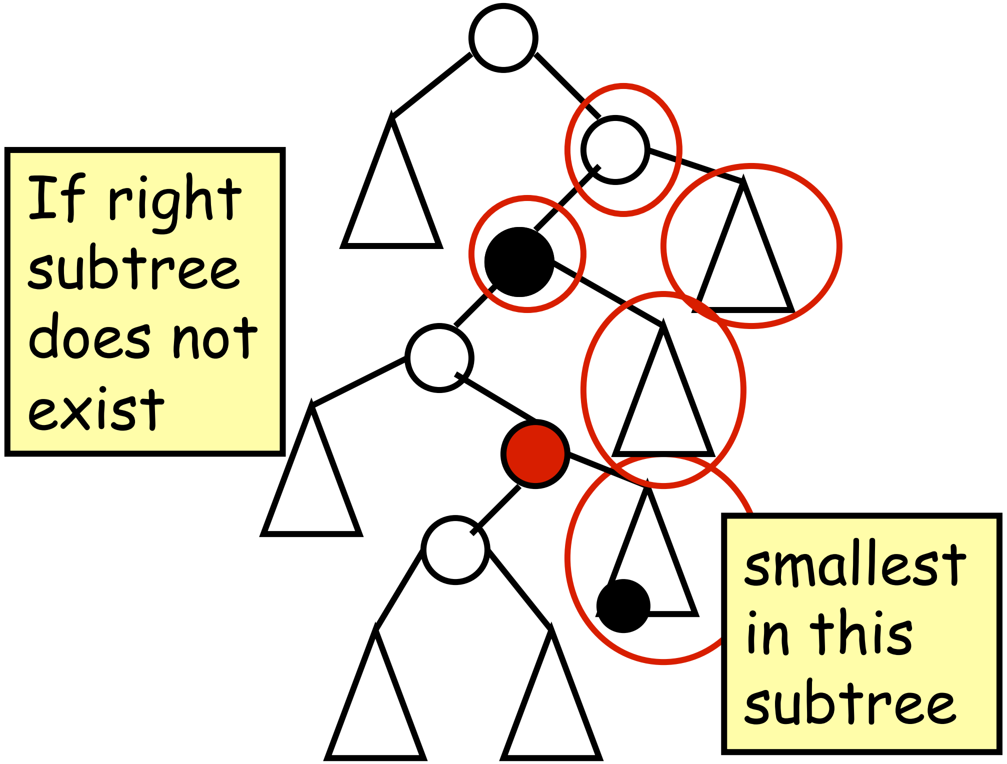
Predecessor(x): return the node whose key is just smaller than that of x or Null if key(x) is the smallest key

Successor

Where are the nodes whose key is larger than yours?

e.g. The node with key = 8
The node with key = 5
The node with key = 4

Among these nodes, which one is the smallest?



If right subtree does not exist

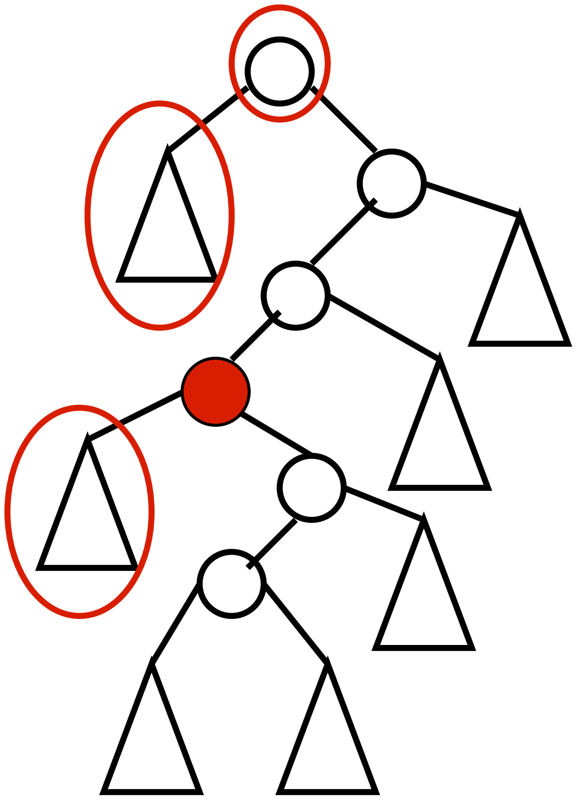
smallest in this subtree

```
Tree-Successor(x) {
    if (x.right $\neq$ null)
        return Tree-Minimum(x.right);
    y = x.p        // parent of x
    while (y $\neq$ null) and (x = y.right) {
        x = y;
        y = y.p;
    }
    return y;
}
```

O(h)

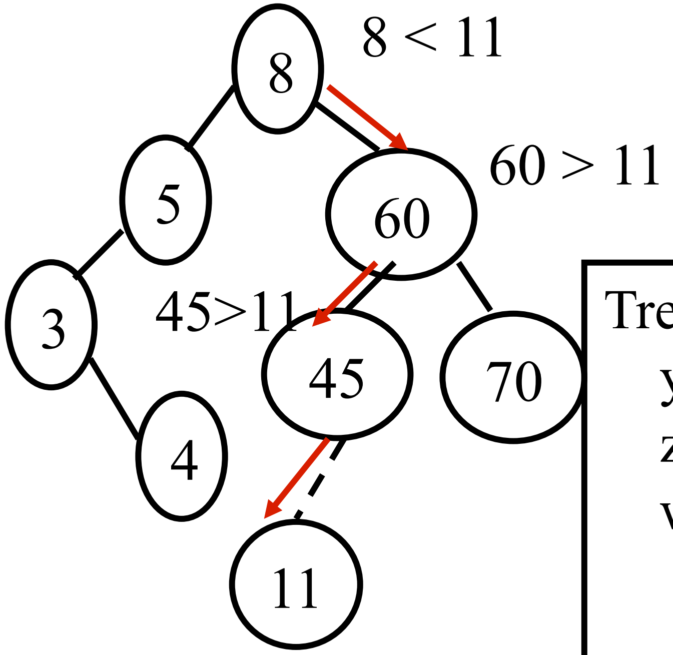
```
Tree-Predecessor(x) {
    if (x.left $\neq$ null)
        return Tree-Maximum(x.left);
    y = x.p        // parent of x
    while (y $\neq$ null) and (x = y.left) {
        x = y;
        y = y.p;
    }
    return y;
}
```

How about Predecessor? Can you write a similar procedure for Predecessor?



Insertion: insert a new element in the tree such that the binary-search-tree property is still maintained

e.g. Insert(11) ~ unsuccessful search



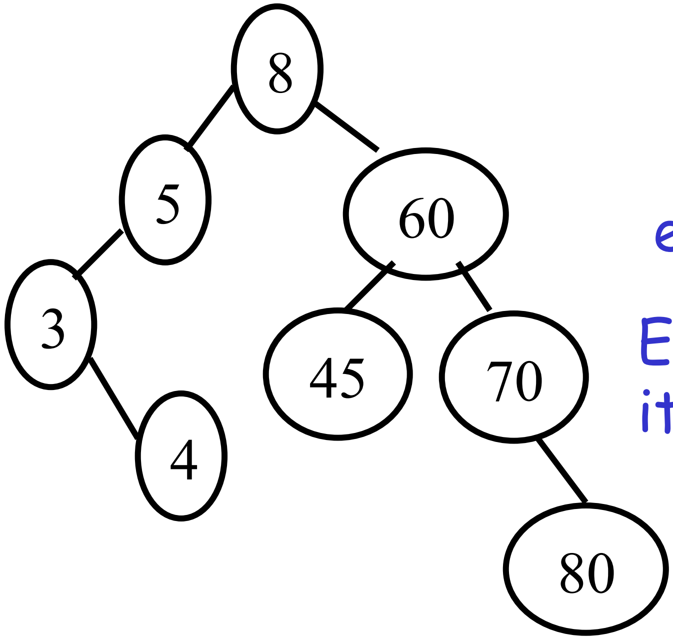
Do not handle the case for null tree

```
if (z = null)
   T = x;
else
   if (x.key....)
```

```
Tree-Insert(T, x){ // x.p=x.left=x.right = null
   y = T;  // Assume T points to the root
   z = null;
   while (y ≠ null) {
        z = y;
        if (x.key < y.key)   // should insert to lefttree
             y = y.left;
        else
             y = y.right;        // should insert to righttree
   }
   x.p = z;
   if (x.key < z.key)    // insert as left child
      z.left = x;
   else                  // insert as right child
      z.right = x;
}
```

O(h)

Deletion: delete an element from T while maintaining the binary-search-tree property.



Yes, we can always reorganize all elements. But can we do better?

e.g. Delete(45); Delete(4); Delete(80)

Easy! Just remove the node as long as it's a leaf

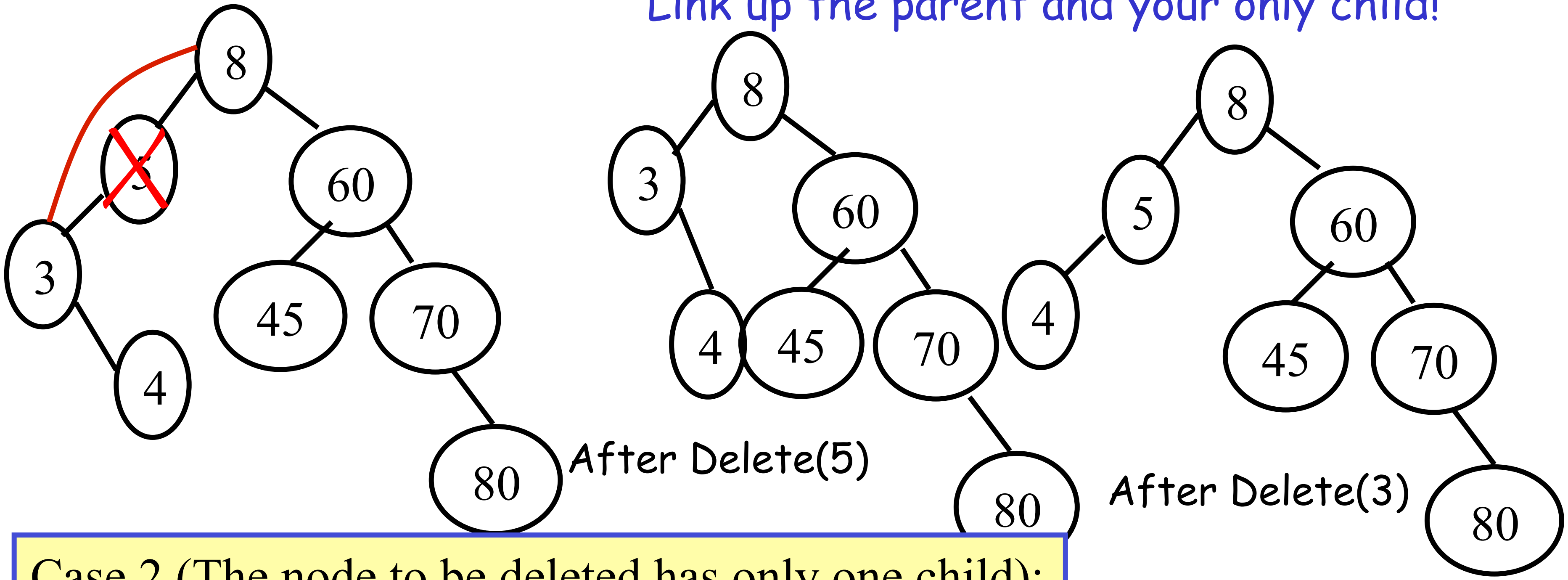
Remark: Do not handle tree with only one node!

Case 1 (The node to be deleted is a leaf):

```
Delete(T, z) { // z points to a leaf to be deleted
    if (z.p.left = z) z.p.left = null;
    else z.p.right = null;
    delete z;  // delete the node and reclaim space
}
```

Deletion: delete an element from T while maintaining the binary-search-tree property. e.g. Delete(5); Delete(3); Delete(70)

Link up the parent and your only child!



After Delete(5)

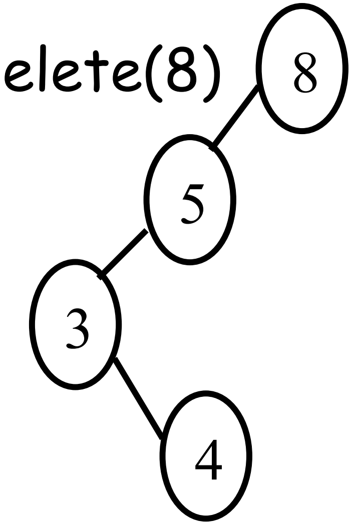
After Delete(3)

Case 2 (The node to be deleted has only one child):

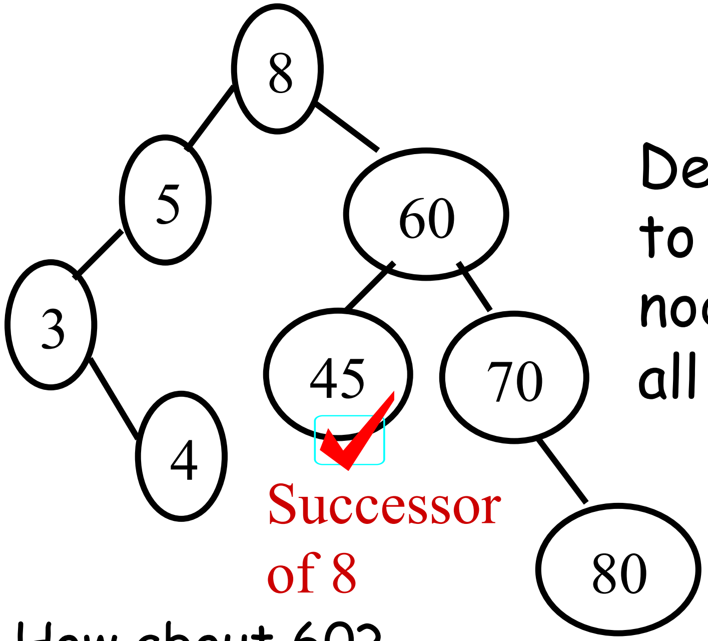
```
Delete(T, z) { // z points to a node to be deleted
    if (z.left = null) x = z.right;
    else x = z.left;
    if (z.p.left = z) z.p.left = x;
    else z.p.right = x;
    x.p = z.p;
    delete z;  // delete the node and reclaim space}
```

Special case: Delete the root

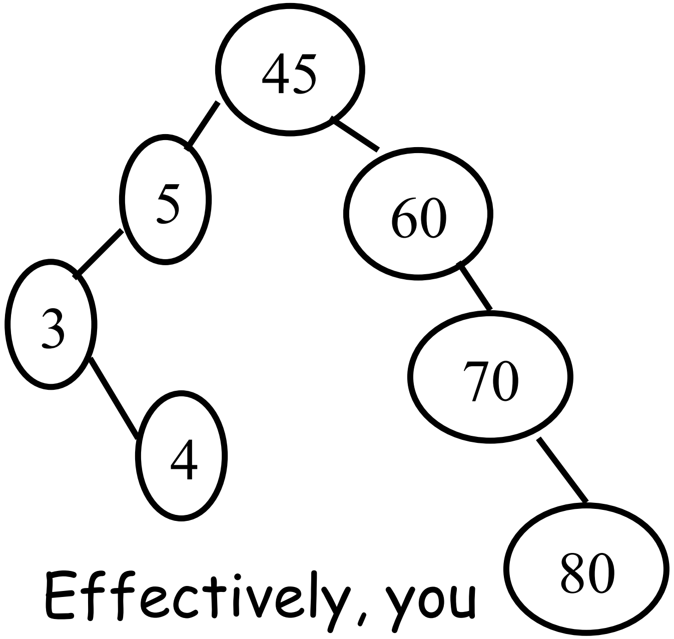
Delete(8)



e.g. Delete(8); Delete(60)

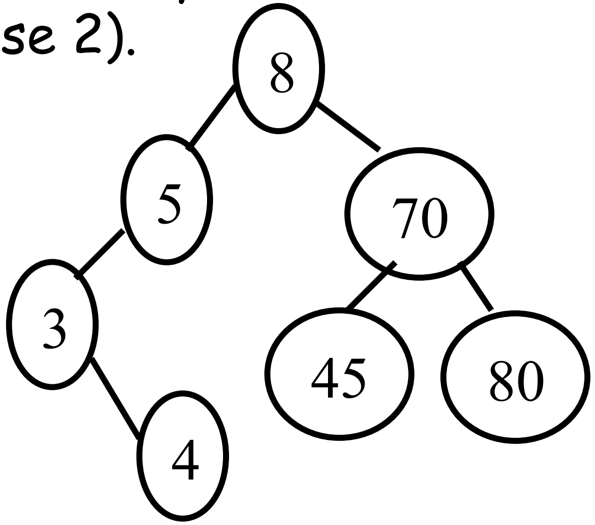


Delete(8): it's easy if we can find someone to replace 8. But it has to be greater all nodes in the left subtree and smaller than all nodes in the right subtree!



Effectively, you replace the content of 8 by 45 and delete the node with 45 (Case 1)

How about 60?
Similarly, the successor of 60 is 70
So, copy 70 to 60 and delete 70.
70 has only one child, so it is easy (Case 2).



Ok, will the successor has two children instead of just one?

No, according to the following [MIT Book 12.2-5, p.293]:
If a node in a binary search tree has two children, its successor has no left child.

<u>Idea of proof:</u>
Let u be the node that has two children and
let v be the node successor of u.
The node v must be in the right subtree of u.
If v has a left child w,
then w is smaller than v, but larger than u.
It contradicts that v is the successor of u.

So, let's summary what we have to do for this case:
Let u be the node to be deleted. Find Successor(u) and let it be v.
Replace content of u by v.
Delete v:

- If v has no child, Case (1)
- If v has one child, Case (2)

Case 3 (The node to be deleted has two children):

```
Delete(T, z) { // z points to a node to be deleted
    x = Tree-Successor(z);
    z.element = x.element;
    if x has no child
        delete x as in Case 1;
    else
        delete x as in Case 2;
}
```

O(h)

In the MIT book, the three cases are combined and handled in one algorithm.

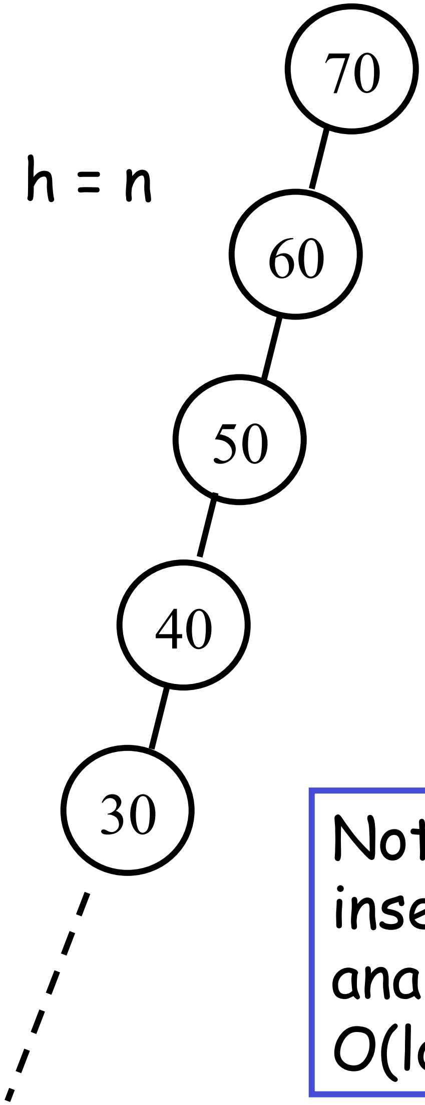
Summary

All the operations Minimum, Maximum, Successor, Predecessor, Search, Insert, Delete run in O(h) time where h is the height of the tree.

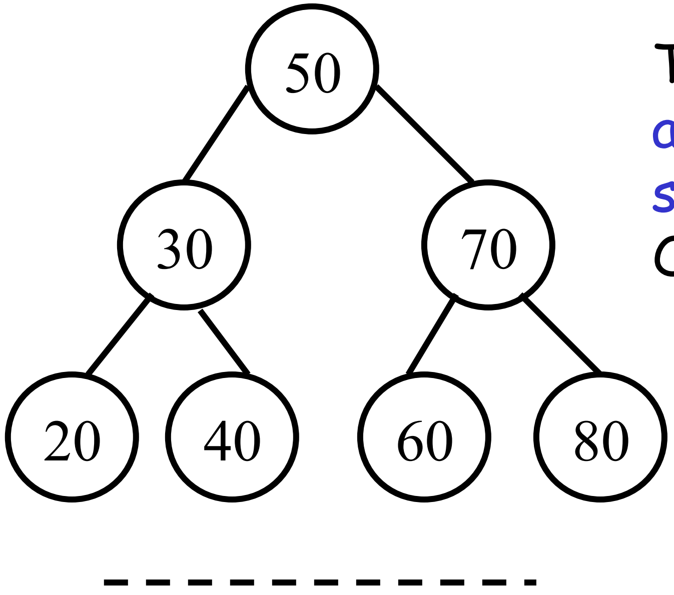
What will be h? (worst case, average case, best case)

Reminder: Worst case

h = n



Best case



h = log n

Average case

The expected height of a randomly built binary search tree on n keys is O(log n)

The tree results from inserting n keys in random order where each of the n! permutations of keys is equally likely

Note: if we allow delete and insert keys randomly, the analysis is difficult and the O(log n) result may not hold

Remark 1

Binary-search-property allows us to get a sorted list of the keys easily from a binary search tree. How?

Note: nodes in your left subtree is smaller than you, and the nodes in your right subtree is greater than you

So.... an inorder tree traversal should print out a sorted list.

```
Inorder-Tree-Walk(x) {
    if (x ≠ null){
        Inorder-Tree-Walk(x.left);
        output(x.key);
        Inorder-Tree-Walk(x.right);
    }
}
```

O(n)

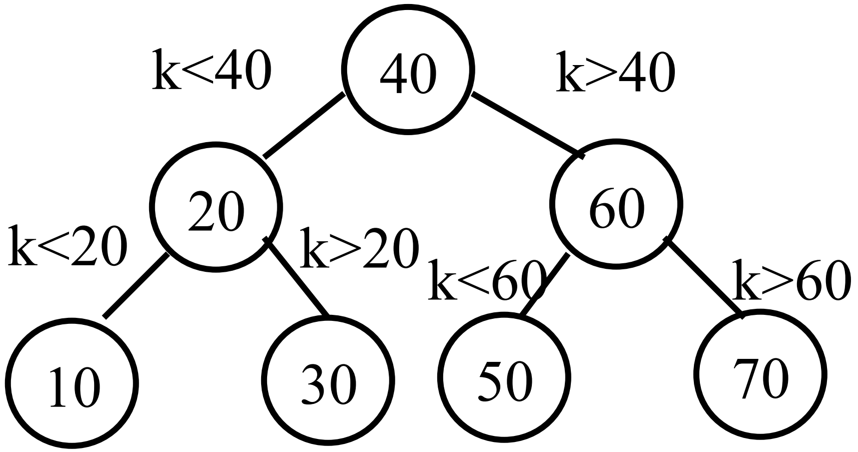
Remark 2

Binary Search and Binary Search Tree

e.g. Perform a binary search on the following sorted sequence:
10, 20, 30, 40, 50, 60, 70

Compare k with 40. If k < 40, check 20, otherwise check 60...

Represent this scheme using a tree:



k<40
40
k>40
20
60
k<20
k>20
k<60
k>60
10
30
50
70