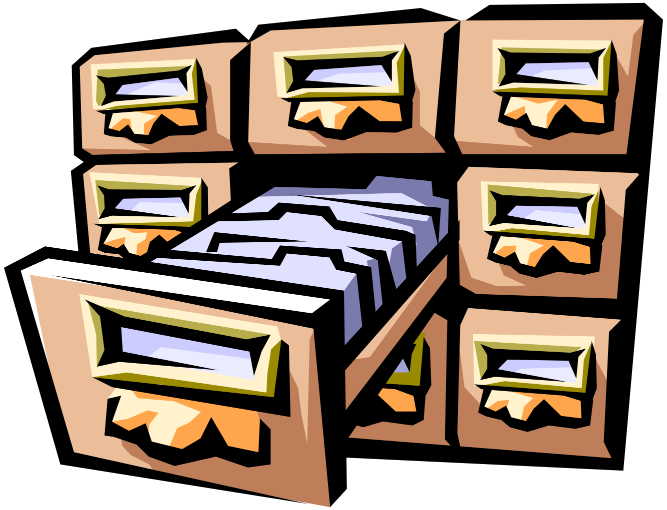
# Maps and Sorted Maps; Binary Search Trees



# Map ADT

A MAP is an ADT to efficiently store and retrieve values based on a uniquely identifying **search key**.

It stores key-value pairs (k,v), which we call **entries**.

Keys are unique/no repeats (they uniquely identify the value); a key is mapped to a value.

The main operations of a MAP are **searching**, **inserting**, and **deleting** items.

Examples: student records, user accounts, etc

Typical keys are username, user ID, etc.

Maps are also know as **associative arrays**.

**Dictionary** ADT is related, although it normally refers to a similar ADT that allows repeated keys.

# The MAP ADT methods:

**get(k):** returns the value v associated to key k, if such entry exists; otherwise returns null.

**put(k, v):** if M does not have an entry with key k, then adds (k,v) and returns null; otherwise it replaces with v the value of the entry with key equal to k and returns the old value.

**remove(k):** removes from M the entry with key k and returns its value; if M has no such entry, then returns null.

**size():** returns the number of entries in M.
**isEmpty():** boolean indicating if M is empty.
**keySet(), values(), entrySet()** returns an iterable collection of keys, values, key-value entries (respectively) stored in M.

# MAP ADT: examples

Applications/examples:

- University information system:

  key= student id

  value= student record (name, address, course grades)

- A domain name system (DNS) maps

a host name (key, e.g. www.wiley.com) to a

a IP address (value, e.g. 208.215.179.146)

- A social media site maps

a username which is the key (usually nonnumeric) to

the user info which is the value (typically tons of personal info)

# SORTED MAP ADT methods:

In addition to the MAP methods:

get(k); put(k, v); remove(k); size(); isEmpty(); keySet(); values(); entrySet()

A SORTED MAP also provides:

firstEntry(), lastEntry(): returns the entry with smallest key, largest key (respectively), or null if the map is empty.

subMap(k1,k2): returns an iterable list with all the entries greater than or equal to k1, but strictly less than k2.

lowerEntry(k), higherEntry(k), floorEntry(k),ceilingEntry(k)
return the entry with, respectively:

the greatest key < k,  the smallest key > k,
the greatest key <= k,  the smallest key => k.

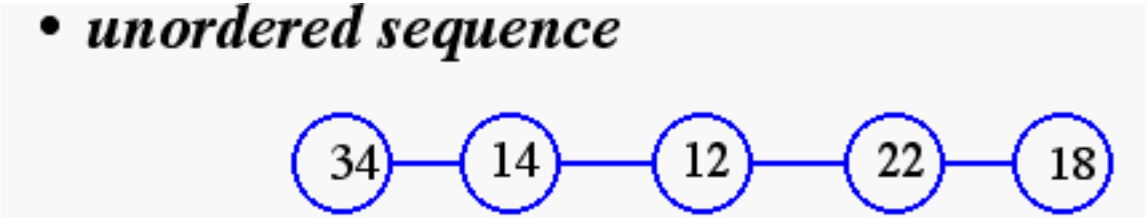
# Implementing MAPs:

- Using an Unordered Sequence
- Using an Ordered Sequence
- Using Search Trees – binary search trees, AVL trees, red-black trees, (2,4)-trees
  (starts this lecture and continue on next ones)
- Using Hash Table
  (discussed at a later lecture)

# Implementing SORTED MAPs:

- Using an Ordered Sequence
- Using Search Trees – binary search trees, AVL trees, red-black trees, (2,4)-trees

# Implementing MAPs with an Unordered Sequence




- get, remove and put takes O(n) time
- The "insert" part takes O(1) time, but we need first to search for key duplicate which takes O(n).

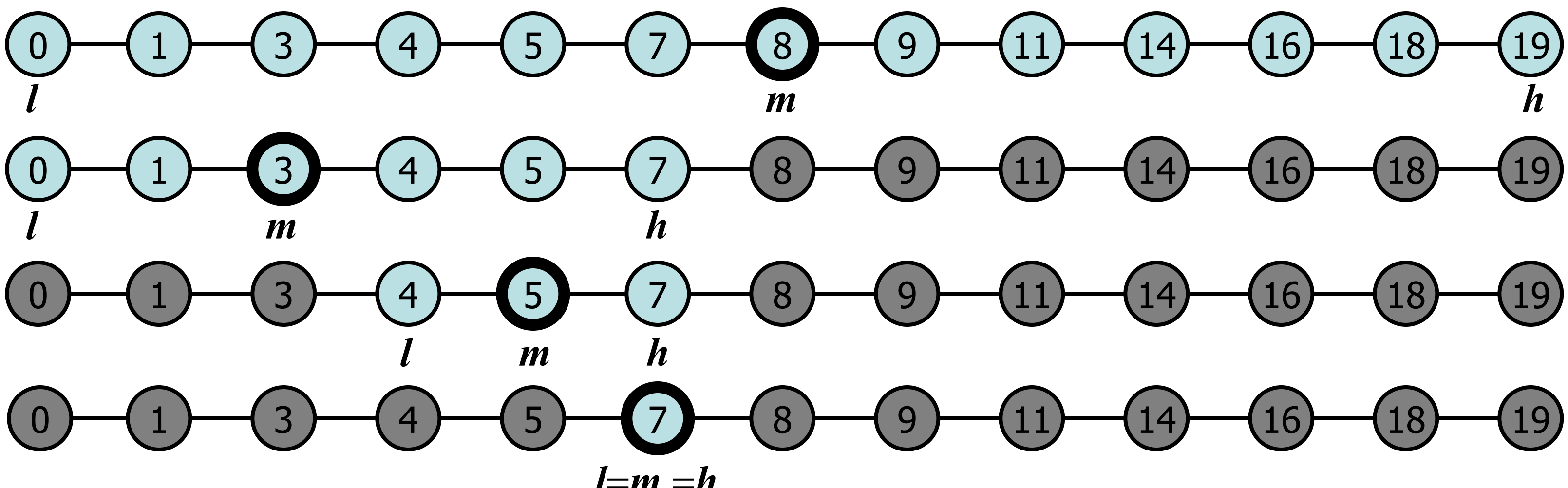
# Implementing a Map an Ordered Sequence

- ***array-based ordered sequence*** (assumes keys can be ordered)


- searching takes O(log n) time (binary search)
- inserting and removing takes O(n) time
- application to look-up tables (frequent searches, rare insertions and removals)

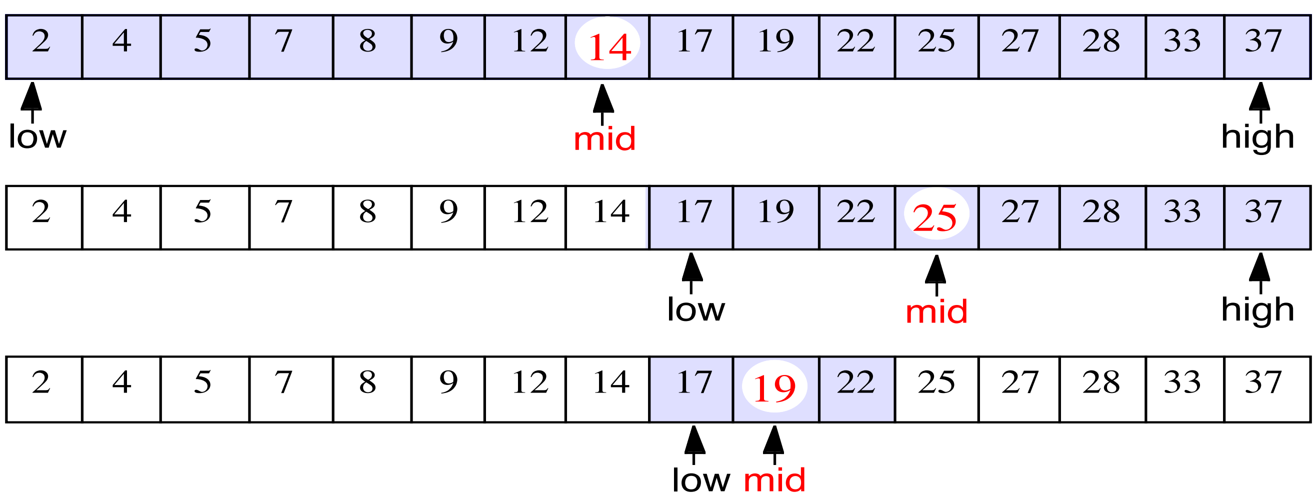
# Binary Search

- narrow down the search range in stages
- "high-low" game
- Example: get(7)



# Pseudocode for Binary Search

```
Algorithm BinarySearch(S, k, low, high)
if low > high then
          return NO_SUCH_KEY
else      mid ← (low+high) / 2
          if k = key(mid)  then
                    return key(mid)
          else if k < key(mid) then
                    return BinarySearch(S, k, low, mid-1)
          else      return BinarySearch(S, k, mid+1, high)
```



# Running Time of Binary Search

- The range of candidate items to be searched is *halved after each comparison*

| comparison | search range |
|---|---|
| 0 | $n$ |
| 1 | $n/2$ |
| 2 | $n/4$ |
| ... | ... |
| $2^i$ | $n/2^i$ |
| $\log_2 n$ | 1 |

In the array-based implementation, access by rank takes O(1) time, thus binary search runs in O(log n) time

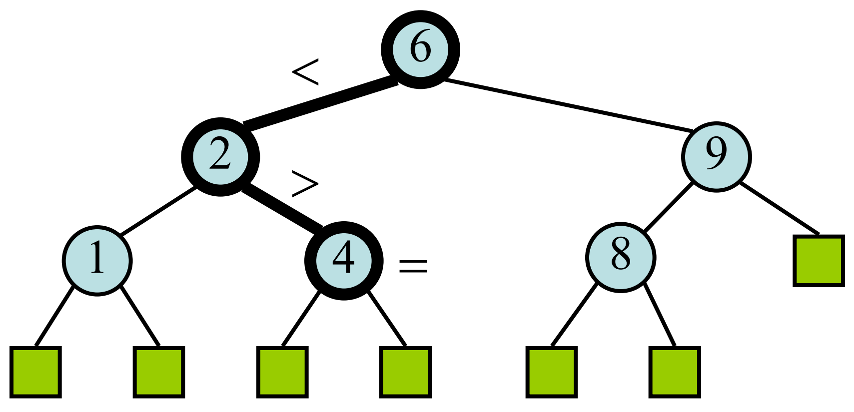
# Binary Search Tree

Searching

Cost of Searching

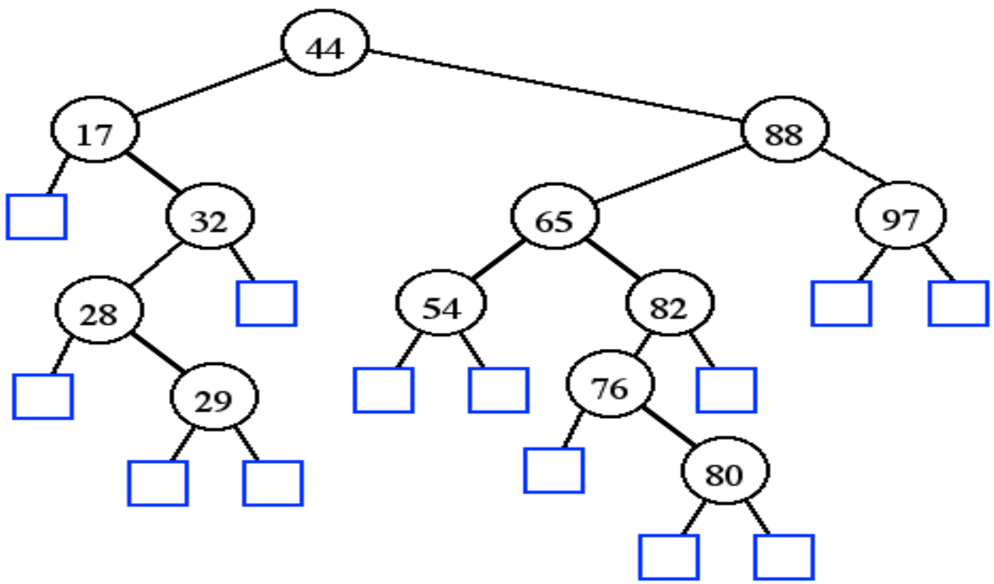
Insertion

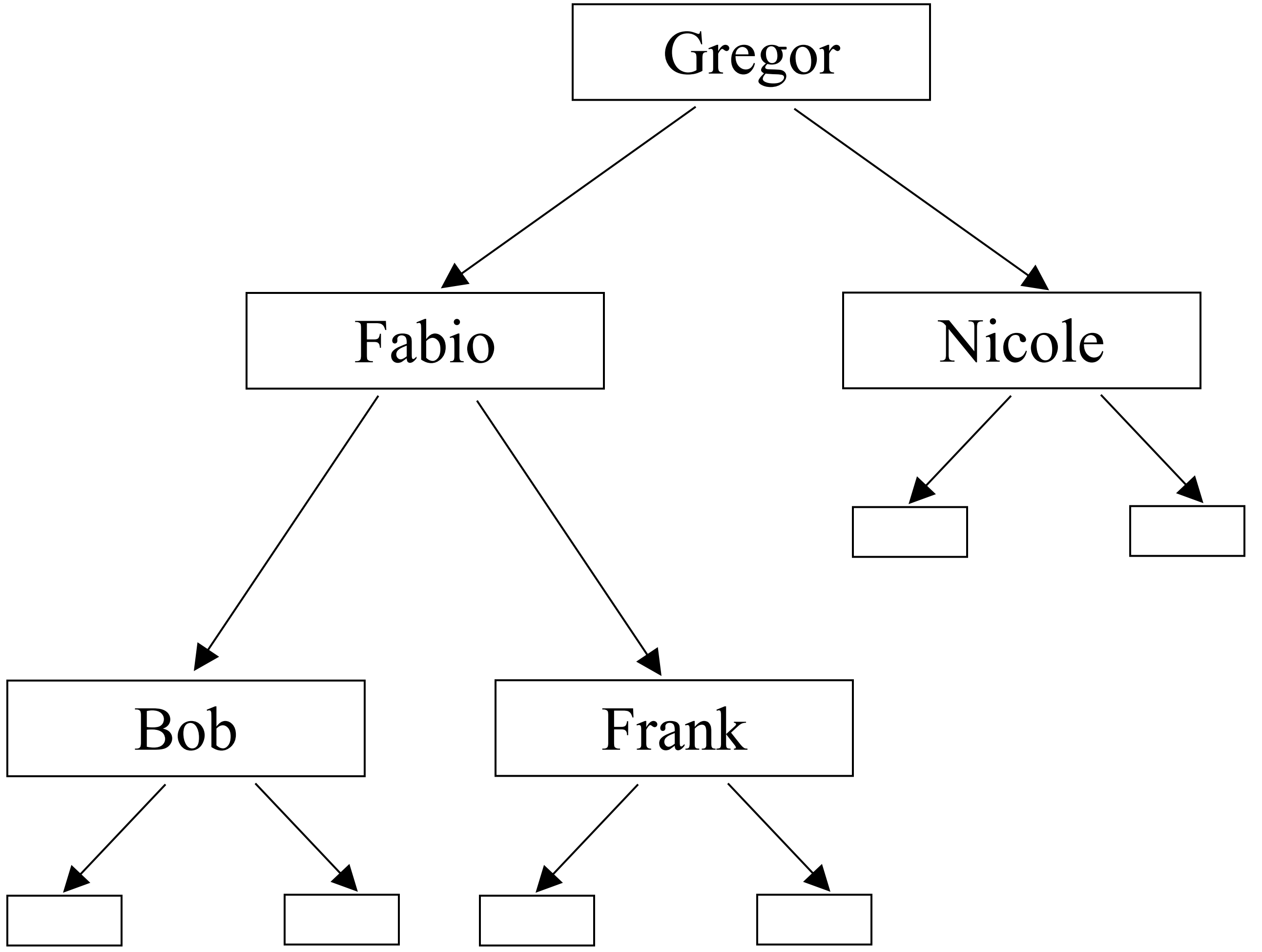
Deletion

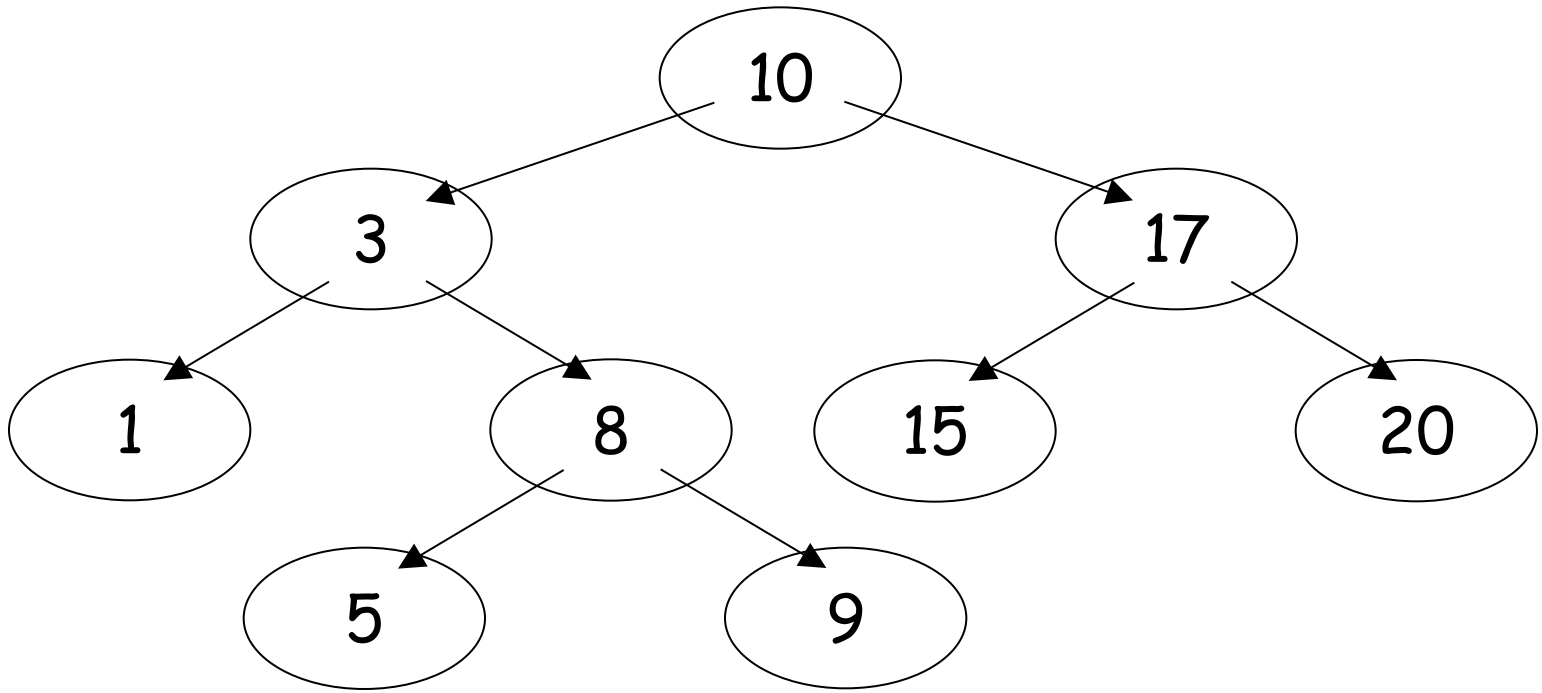


# Binary Search Trees

- A binary search tree is a binary tree T such that
  - each internal node p stores an item (k, v) of a MAP.
  - keys stored at nodes in the left subtree of p are less than k.
  - keys stored at nodes in the right subtree of p are greater than k.
  - external nodes do not hold elements but serve as place holders (dummy leaves).




Gregor
Fabio
Nicole
Bob
Frank




Question: How can we traverse the tree so that we visit the elements in increasing key order?

IN-ORDER TRAVERSAL

always traverses the keys in increasing order

in a binary search tree !!!

# MAP Operations using Binary Search Trees

Searching get(k):

use TreeSearch(k)

Inserting/updating value put(k, v):

use TreeInsert(k,v)

Removing remove(k):

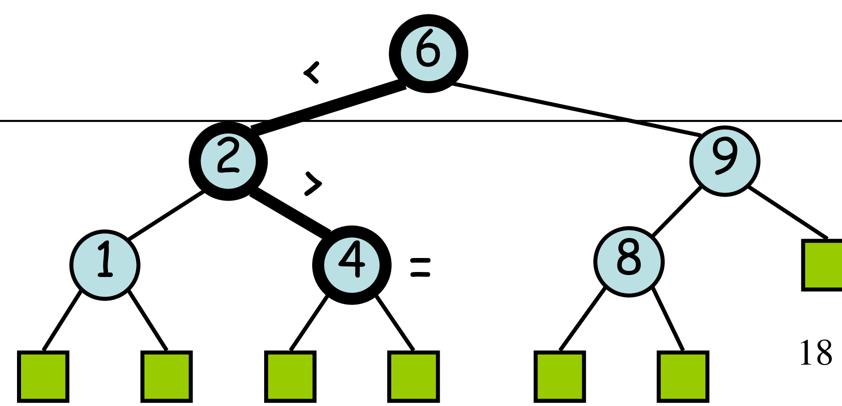
useTreeDelete(k)

# Search

- To search for a key *k*, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of *k* with the key of the current node
- If we reach a leaf, the key is not found and we return this dummy leaf which will help with insertion if needed.
- Example: TreeSearch(4)

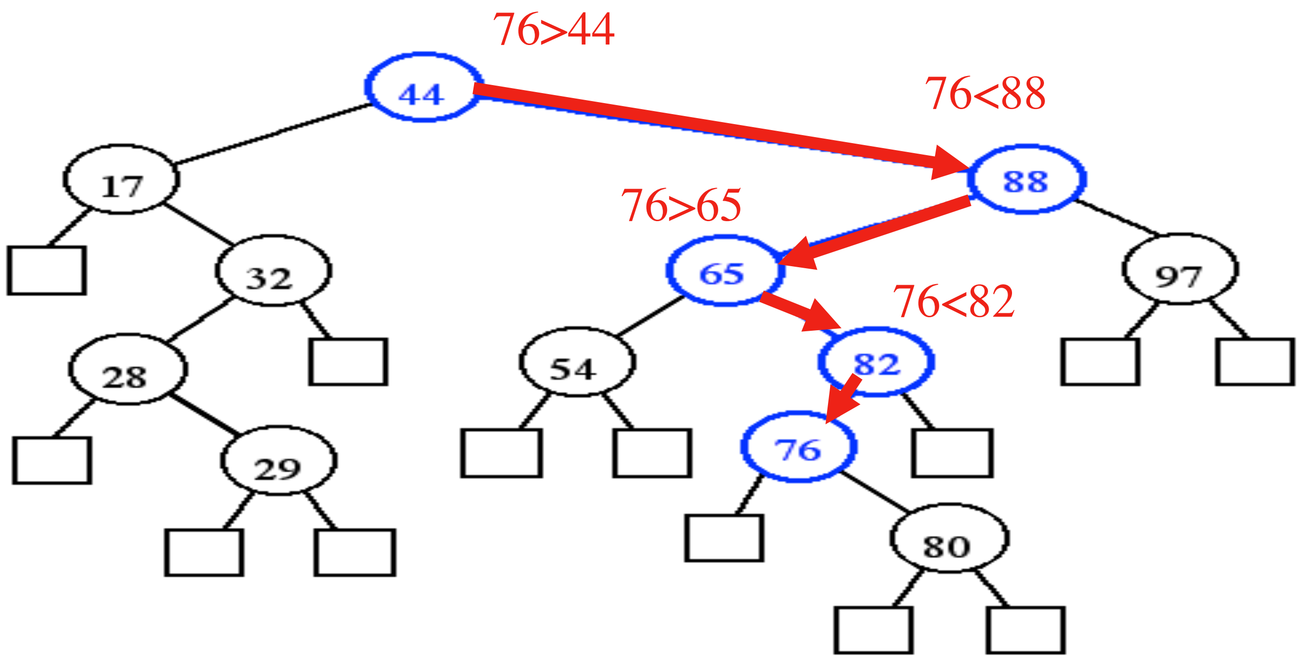
```
Algorithm TreeSearch(k)
    TreeSearch(root,k)

Procedure TreeSearch(p, k)
    if p is external then
        return p {unsuccessful search}
    else if k == key(p)
        return p {successful search}
    else if k < key(p)
        return TreeSearch(left(p),k)
    else { k > key(v) }
        return TreeSearch(right(p),k)
```



# Search Example I

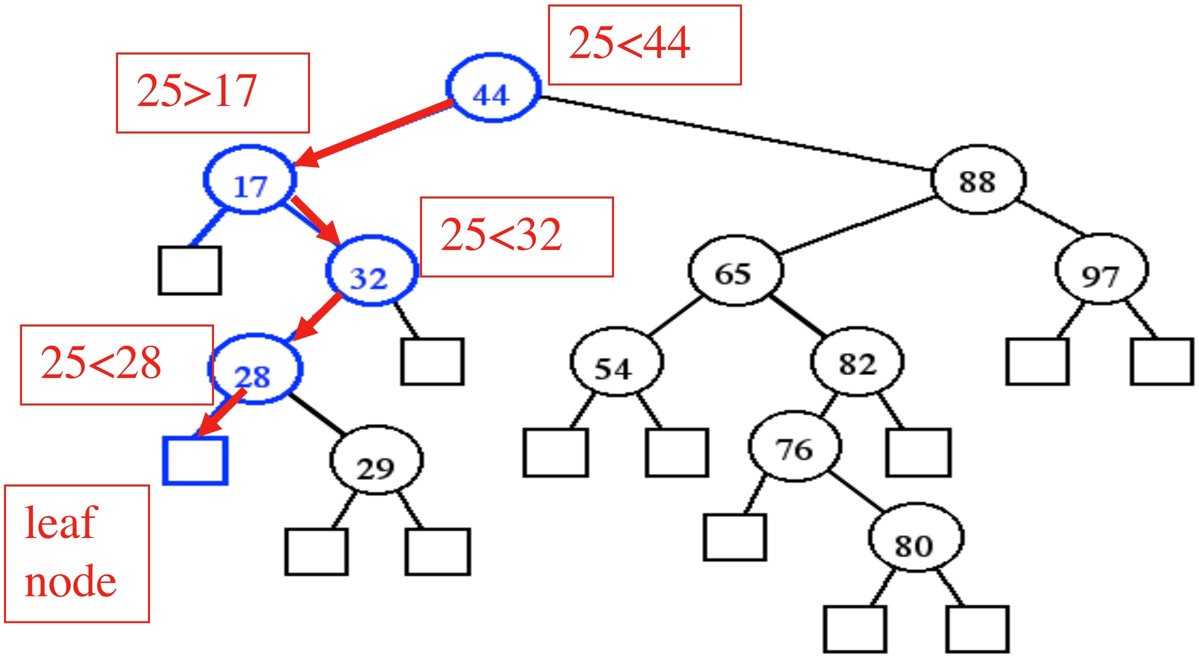
Successful TreeSearch(76)




- A successful search traverses a path starting at the root and ending at an internal node.

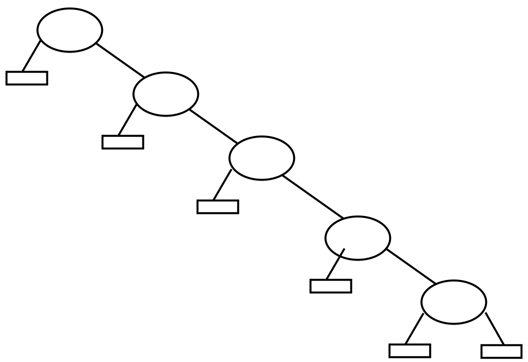
# Search Example II

Unsuccessful TreeSearch(25)

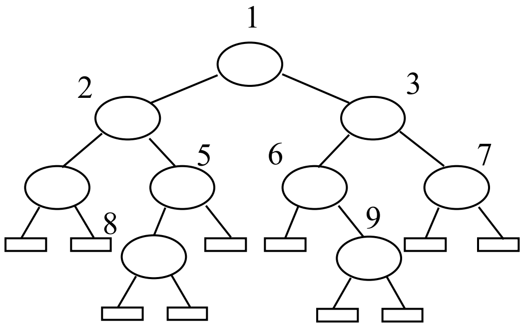



- An unsuccessful search traverses a path starting at the root and ending at an external node

# Cost of Search

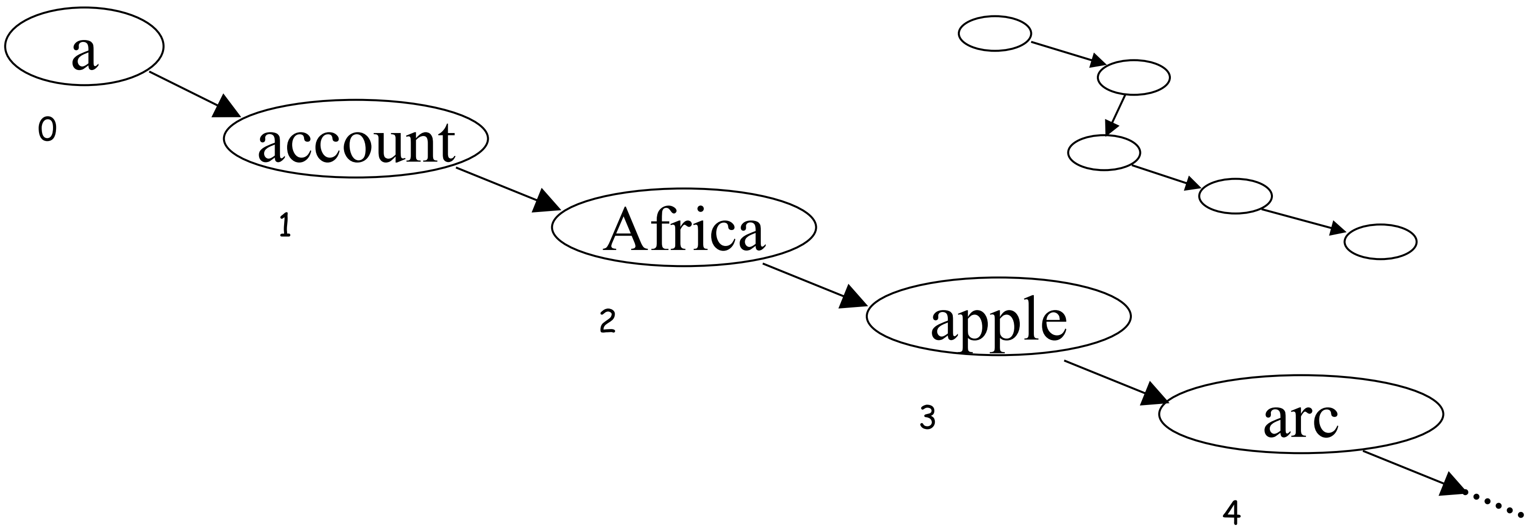


Worst tree




Best tree

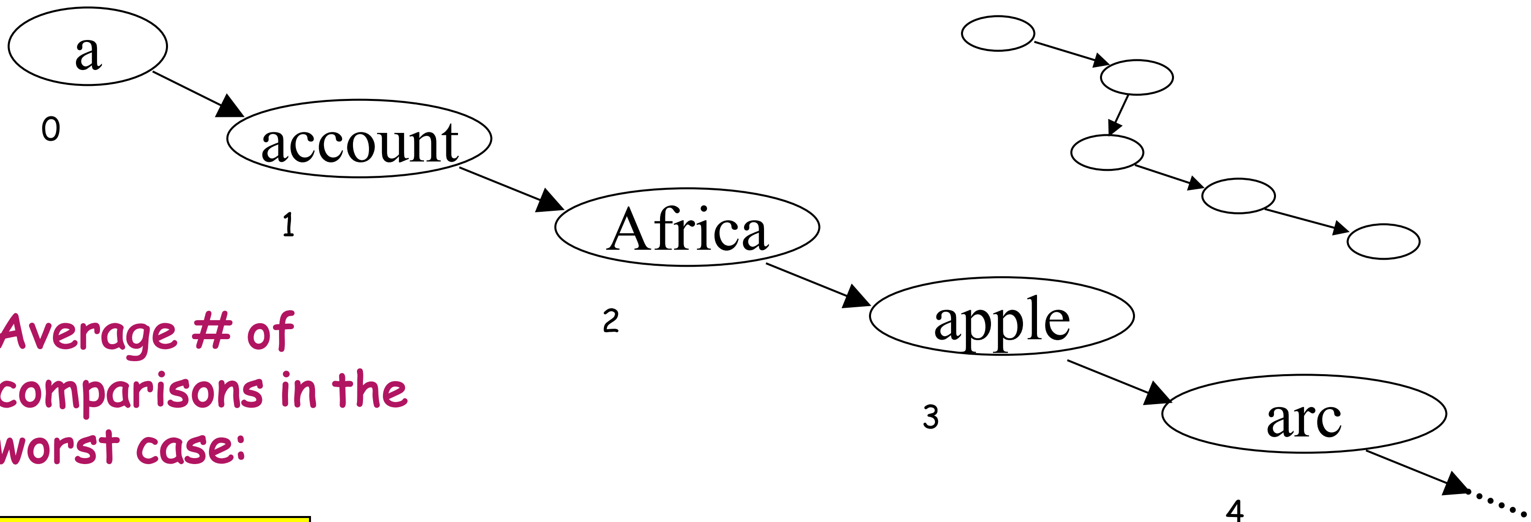
# Cost of Search: Worst Case




Worst possible Tree:
Worst Case:

O(n)

# Cost of Search - Average Worst Case



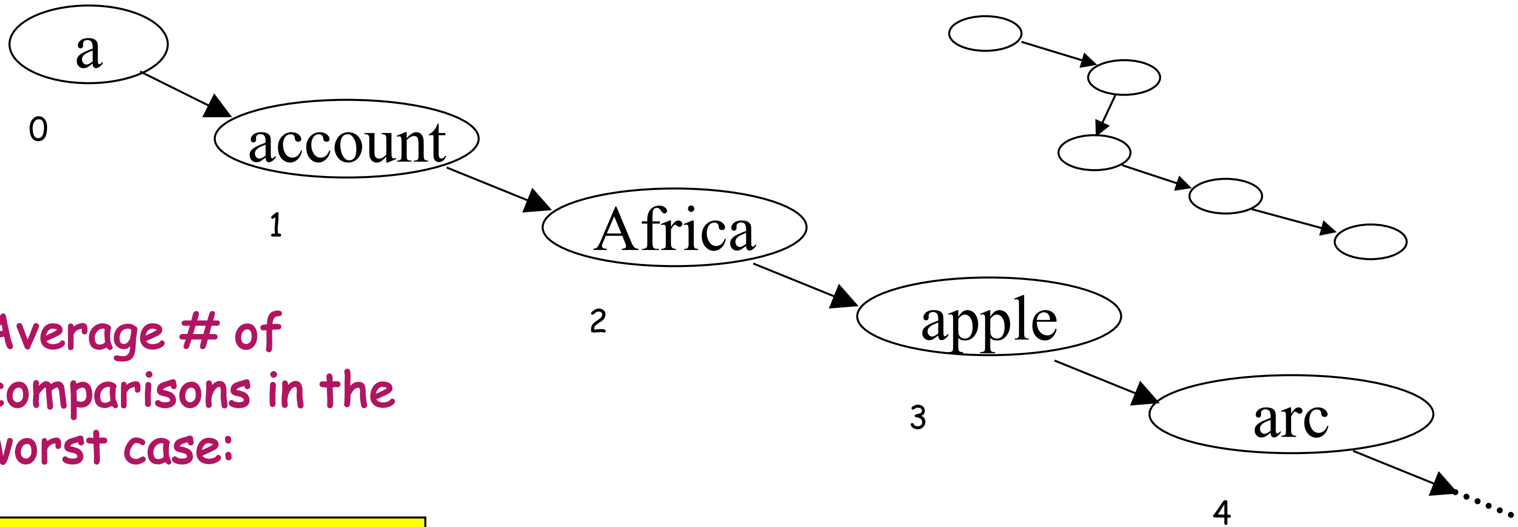

Average # of comparisons in the worst case:

**Successful search**

Path to node i has length i, to get there we do i comparisons

Avg cost= $(1/n)\sum i = O(n)$

# Cost of Search - Average Worst Case

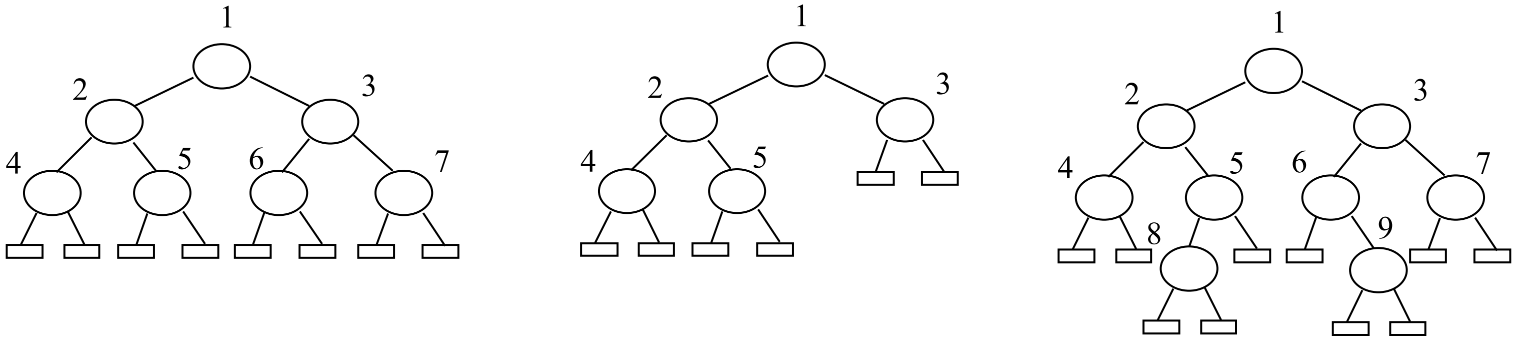



Average # of comparisons in the worst case:

Unsuccessful search

An unsuccessful search always takes O(n) comparisons for n internal nodes

# Cost of Search: Best Case

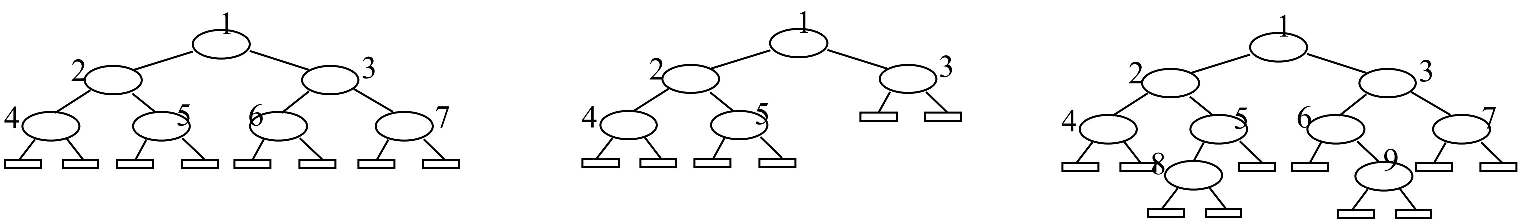



Leaves are on the same level or on an adjacent level.

Length of path from root to node i = $\lfloor \log i \rfloor$

Worst case of the best possible tree: O(log n)

# Cost of Search: Average Best Case




Leaves are on the same level or on an adjacent level.

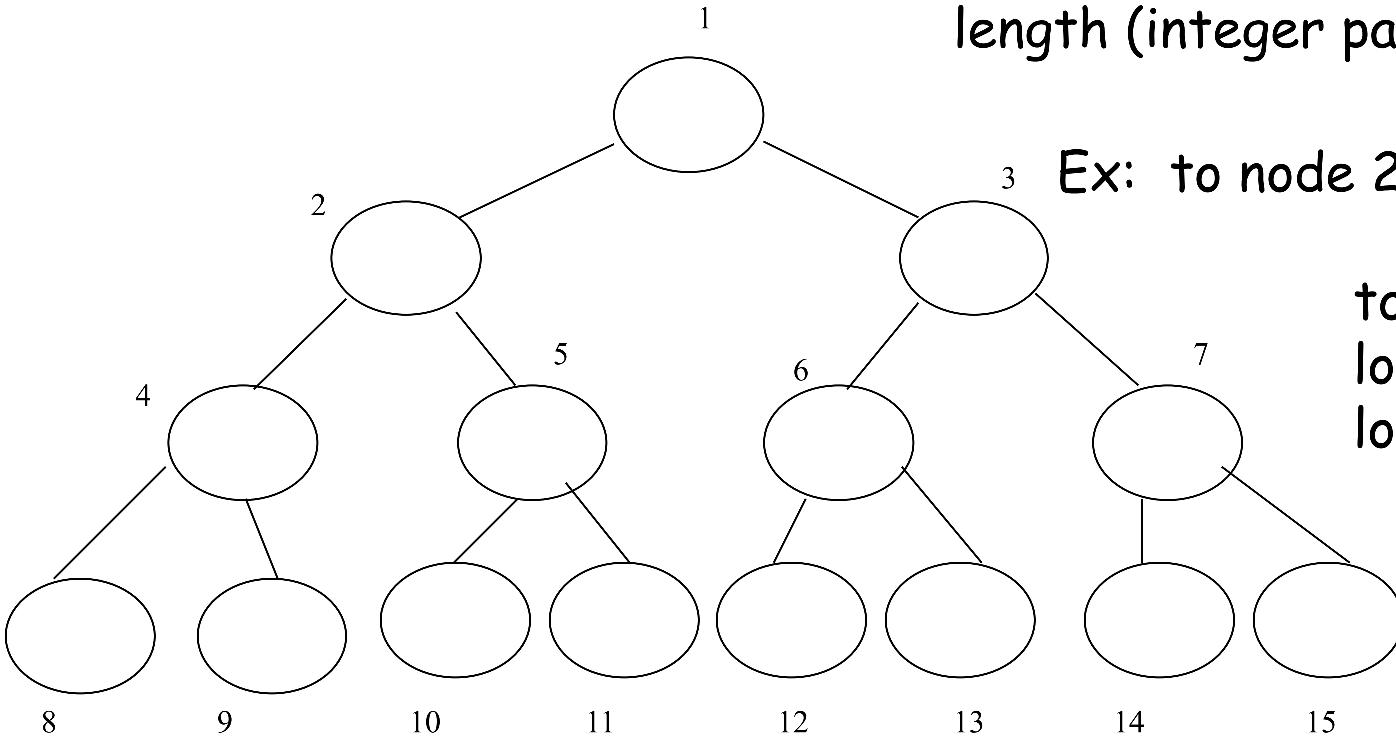
Length of path from root to node i = $\lfloor \log i \rfloor$

Comparisons to node i: log i

Successful search

→ **Average # of comparisons in the best possible tree**

$$\frac{1}{n}\sum_{i=1}^{n} \log i \quad = \quad O((n \log n)/\ n) = O(\log n)$$



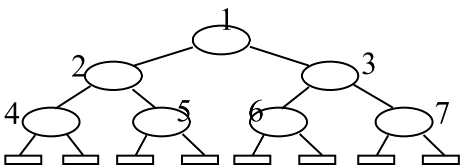

Comparisons to node i: path of length (integer part of) log i

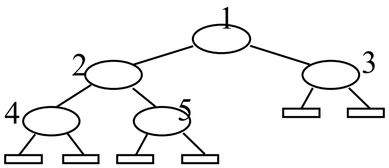
Ex: to node 2: log 2=1

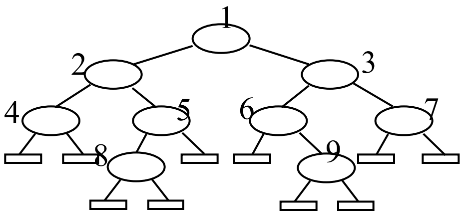
to node 4,5,6,7:
log 4 = log 5=
log 6 =log 7 =2

Comparisons to node i: O(log i)

# Cost of Search: Average Best Case




Leaves are on the same level or on an adjacent level.

Length of path from root to node i = $\lfloor \log i \rfloor$

Only paths to external nodes count.

Unsuccessful search

always O(log n)

# Summary

Worst tree:

Successful search:
- Worst case: O(n)
- Average case: O(n)

Unsuccessful search: Always: O(n)

Best Tree:

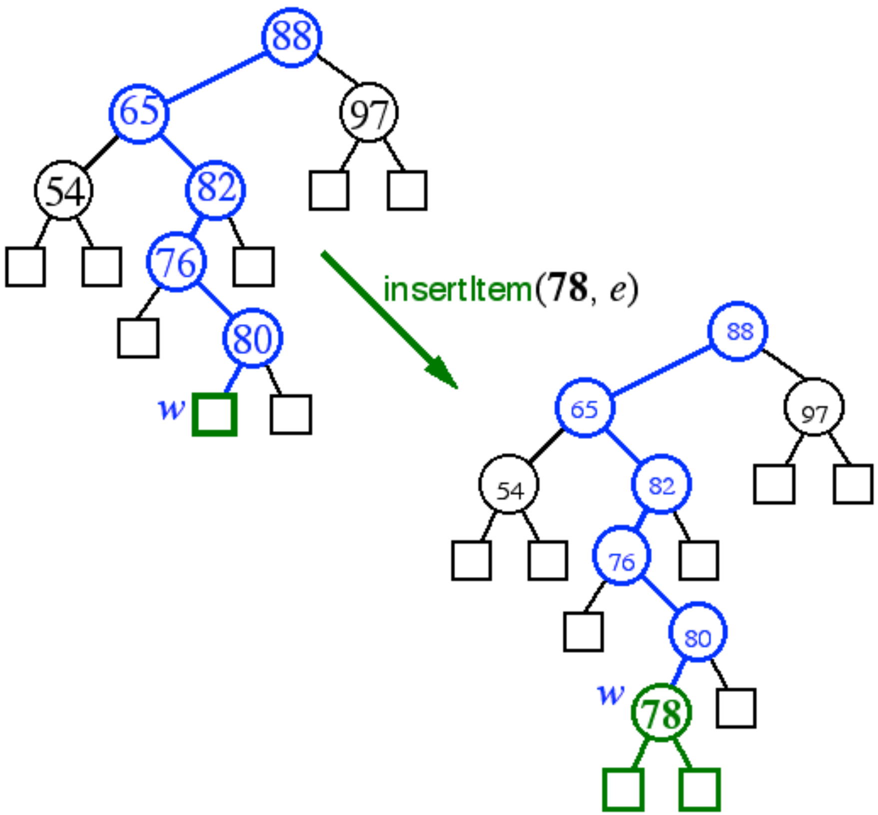
Successful search:
- Worst case: O(log n)
- Average case: O(log n)

Unsuccessful search: Always: O(log n)

**Note: worst case search for arbitrary binary search tree O(n)**

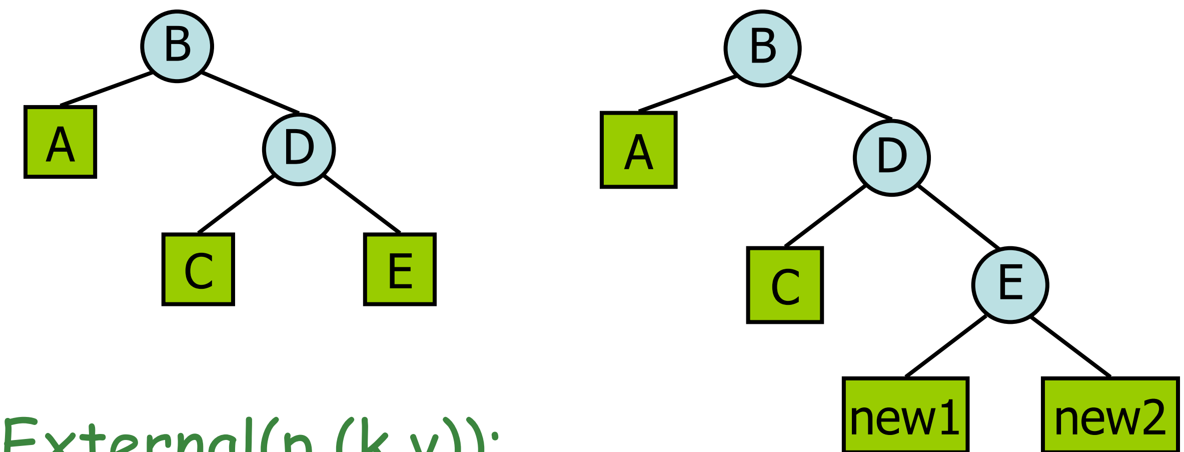
# Insertion case I

- To perform TreeInsert(k, v), let w be the node returned by TreeSearch(k, T.root())
- If w is external, we know that k is not stored in T. We call expandExternal(w, (k,v)) to store (k, e) in w



# expandExternal(p,(k,v)):

Transform p from an external
node into an internal node by creating two new children




expandExternal(p,(k,v)):
if isExternal(p)
    create new nodes new1 and new 2
    p.left $\leftarrow$ new1
    p.right $\leftarrow$ new2
    store entry (k,v) in p
    size $\leftarrow$ size +2

## Insertion case II

- If w is internal, we know the item with key k is stored at w.

In this case, we just replace the value on this node to the given value v.

# Insertion in a Binary Search Tree

```
Algorithm TreeInsert(k,v)
        p = TreeSearch(root(),k)
        if k == key(p) then
            change p's value to (v)
        else
           expandExternal(p,(k,v)):
```

# Construct a Tree

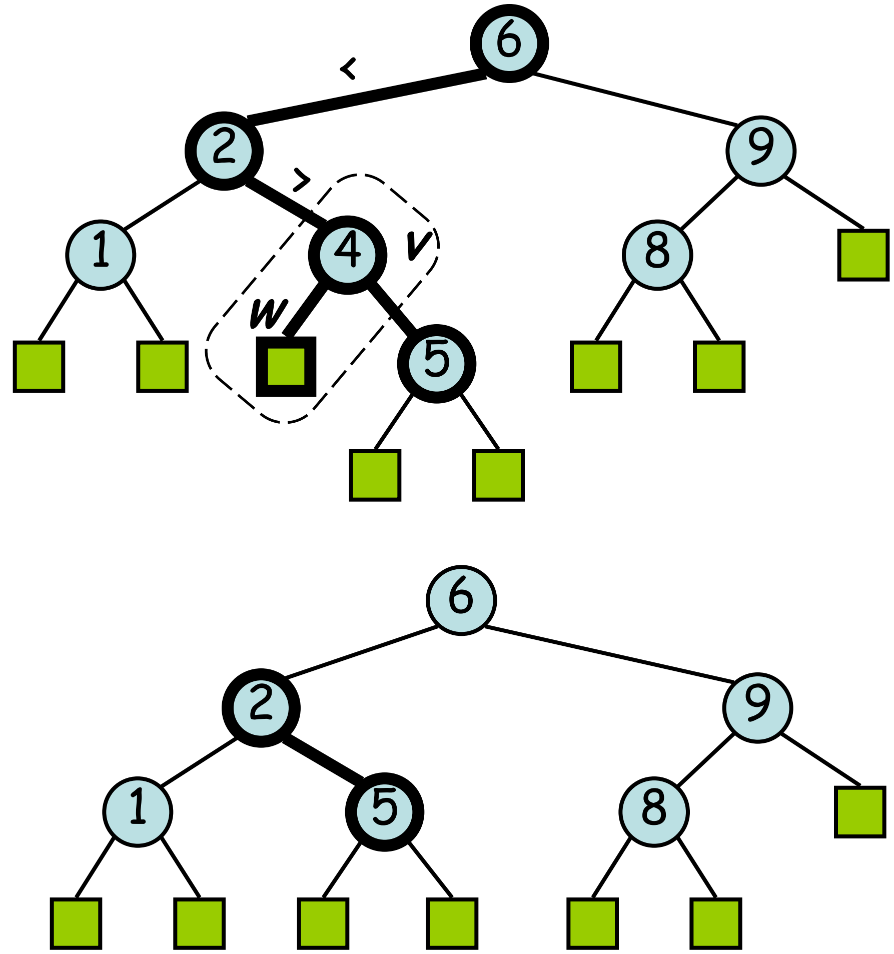
What would be the result of constructing a tree from repeated insertions of the following sequences?

a. 5,8,3,7,1,9,2,4,6
b. 1,2,3,4,5,6,7,8,9
c. 5,4,6,3,7,2,8,1,9

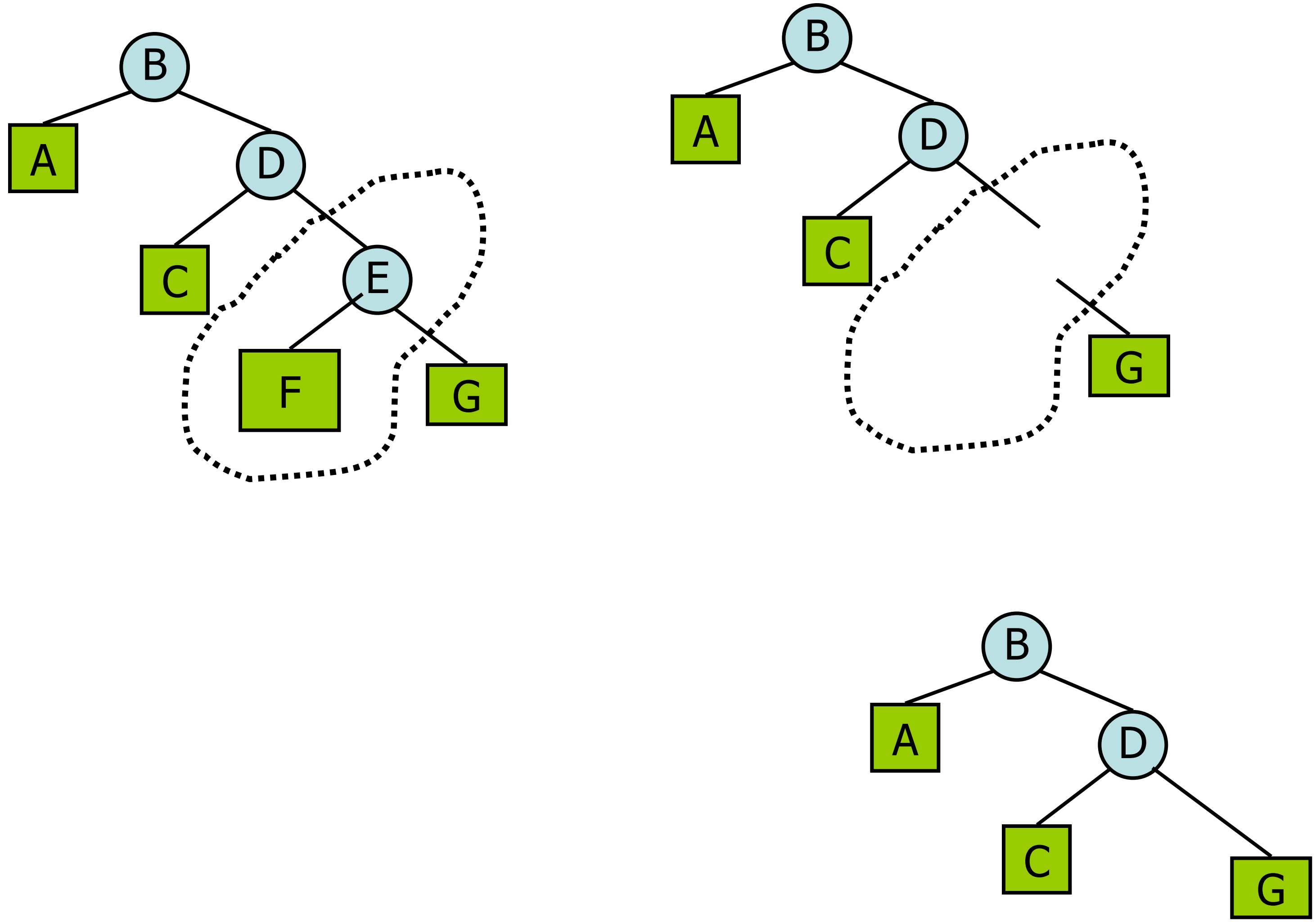
When do you think trees work best?

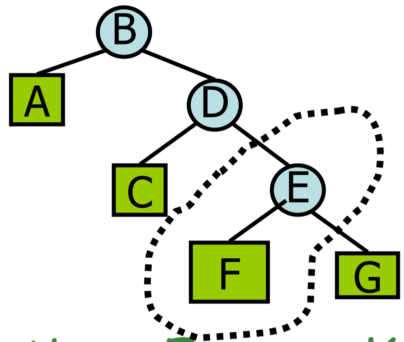
# Deletion I

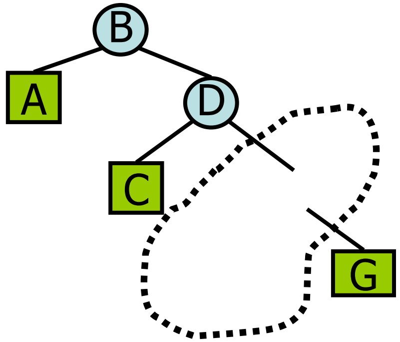
- To perform operation remove(***k***), we search for key ***k***
- Assume key ***k*** is in the tree, and let ***v*** be the node storing ***k***
- If node ***v*** has a leaf child ***w***, we remove ***v*** and ***w*** from the tree with operation removeAboveExternal(***w***)
- Example: remove 4

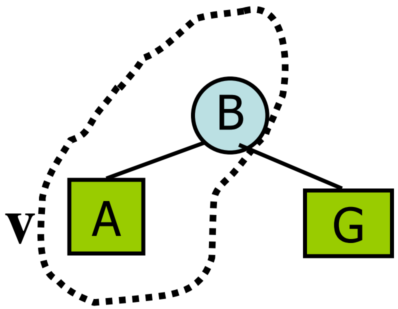


# removeAboveExternal(v):

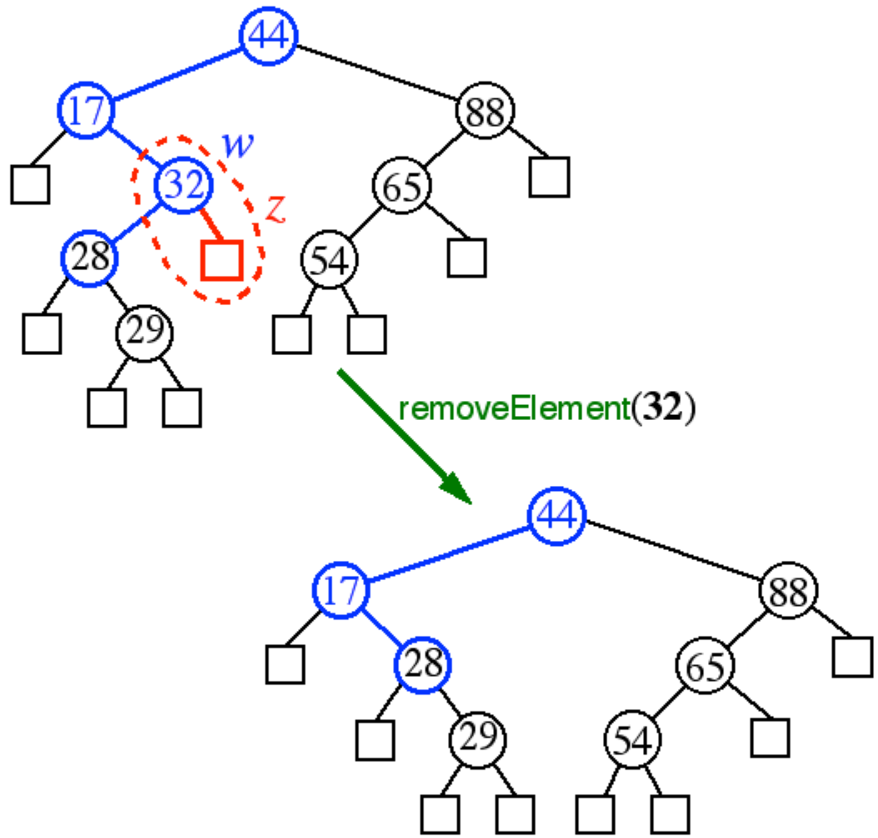




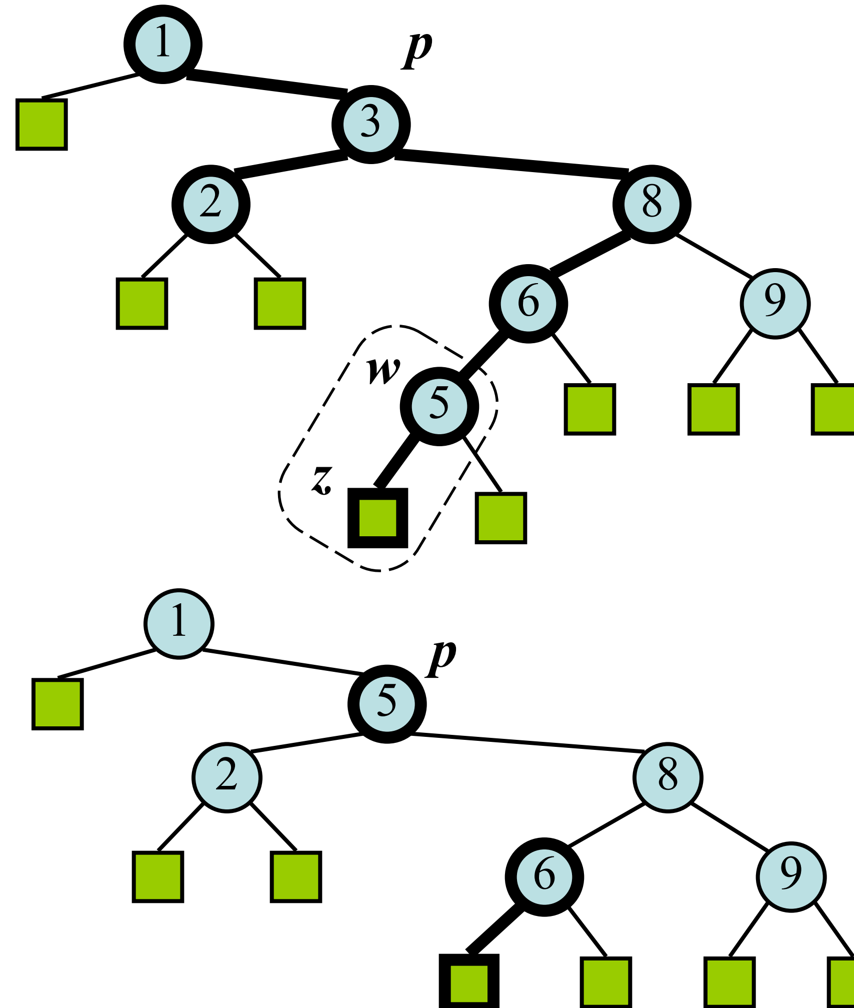




```
removeAboveExternal(v):
 if isExternal(v)   {
    p ← parent(v)
    s ← sibling(v)
    if isRoot(p)  {
        s.parent   ←  null
        root ← s
    }
    else {
         g ← parent(p)
         if (p is leftChild(g)   g.left ← s
               else g.right ← s
         s.parent ← g
         }
    size ← size - 2
 }
```

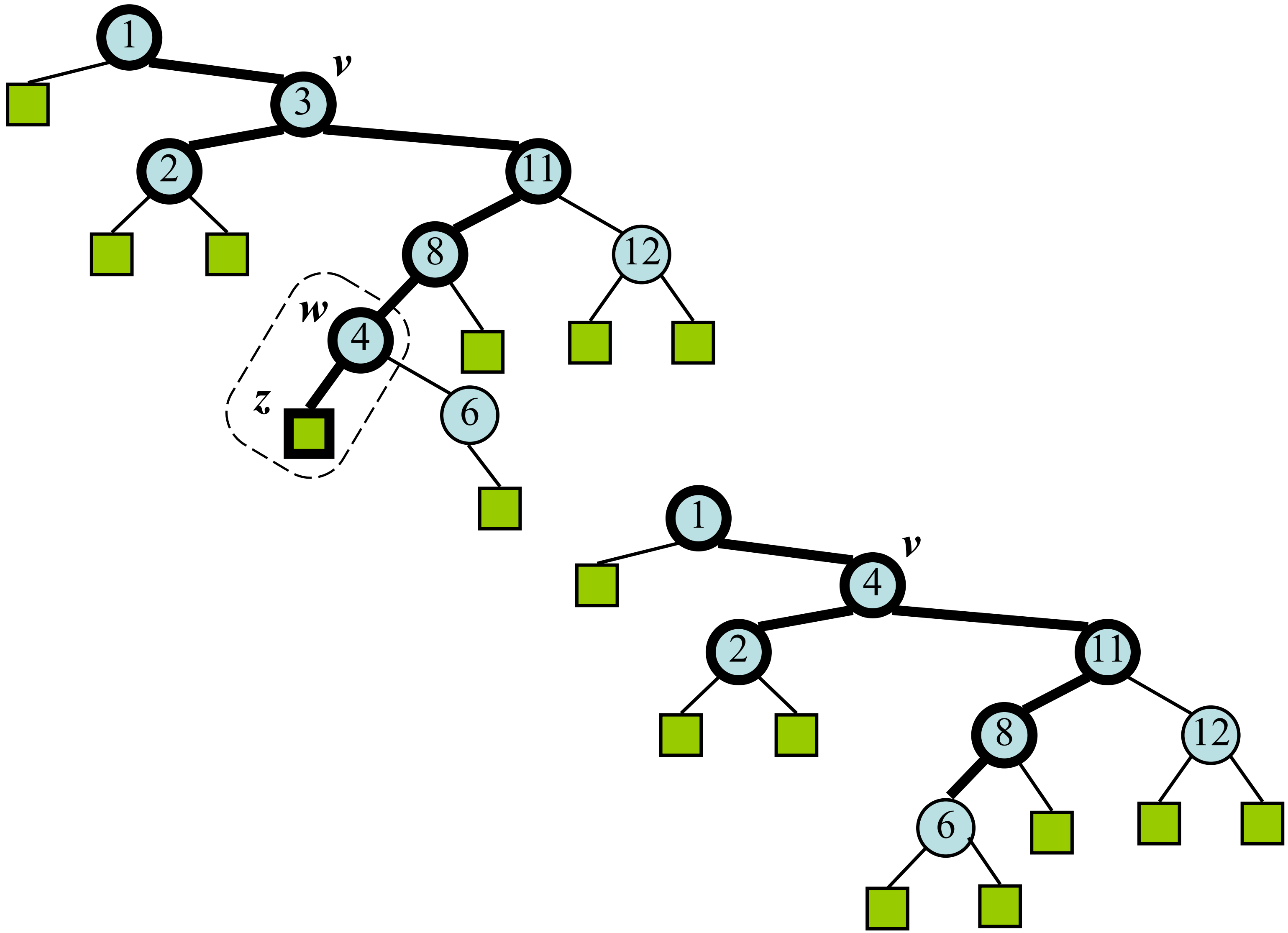

44
17
88
w
32
65
z
28
54
29
removeElement(32)
44
17
88
28
65
29
54

# Deletion II

- We consider the case where the key *k* to be removed is stored at a node *p* whose children are both internal
  - we find the internal node *w* that follows *p* in an inorder traversal (note w it does not have a left child!)
  - we copy *entry*(*w*) into node *p*
  - we remove node *w* and its left child *z* (which must be a leaf) by means of operation removeAboveExternal(*z*)
- Example: remove(3)




Note: see textbook for different approach: locating node w that preceeds p in inorder traversal. How would this change the method above?


1
v
3
2
11
8
12
w
4
z
6
1
v
4
2
11
8
12
6

# Practice, practice, practice…

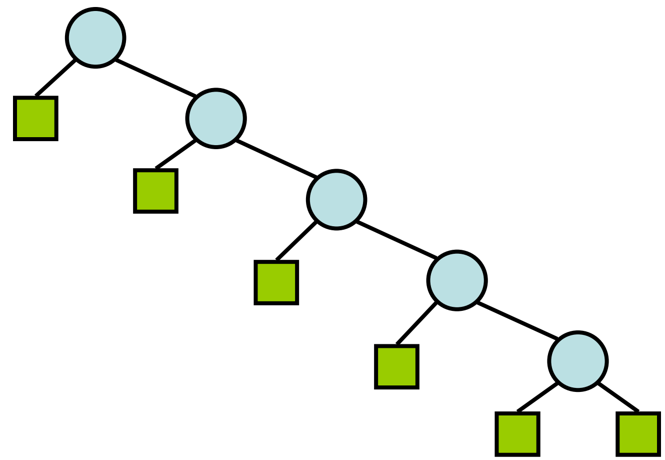
a. Delete the 3 from the tree you got in exercise (a) in page 33.
b. Now delete node 5.

# Cost of Inserting and Deleting = Cost of Search

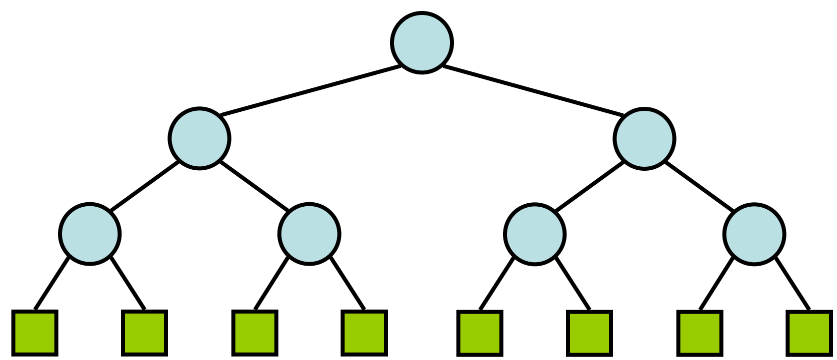
**Summary:**

Consider a dictionary with $n$ items implemented by means of a binary search tree of height $h$

- the space used is $O(n)$
- methods findElement , insertItem and removeElement take $O(h)$ time



The height $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case



# Conclusion

- To achieve good running time,
  we need to keep the tree *balanced*,
  i.e., with O(log n) height.

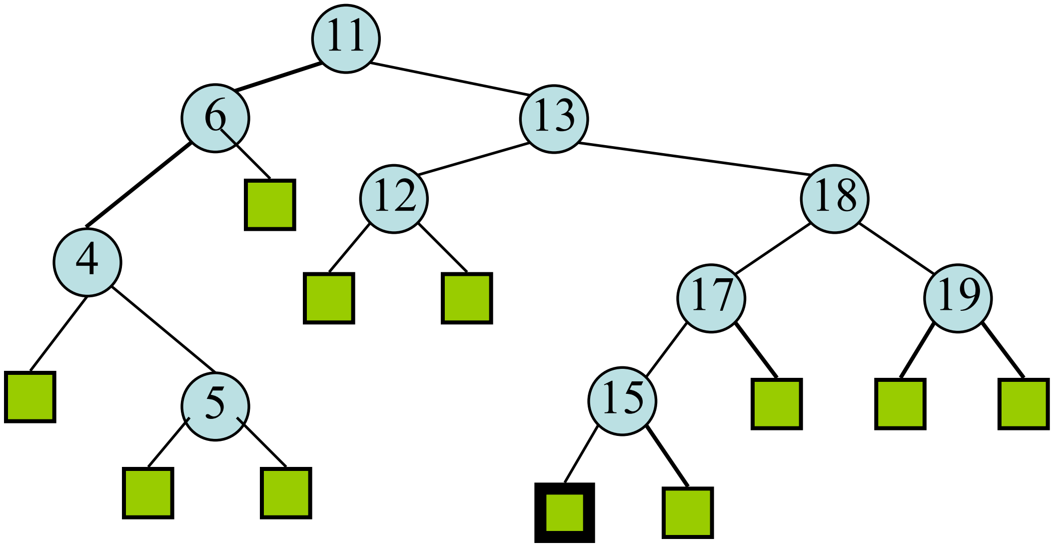
- **Various balancing schemes can be explored:**

AVL trees and red-black trees are a balanced binary search trees: their height is O(log n)

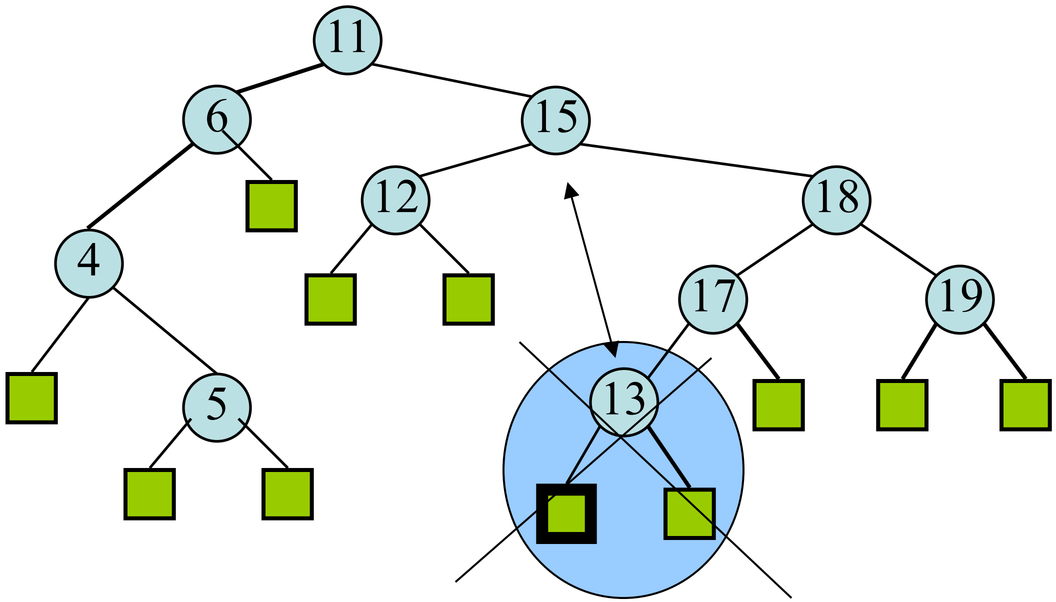
A (2,4)-tree is a search tree (not binary, each internal node has 2, 3 or 4 kids); its height is also O(log n).

Using simply a binary search tree gives worst case running time O(n) for search, insert and delete operations!

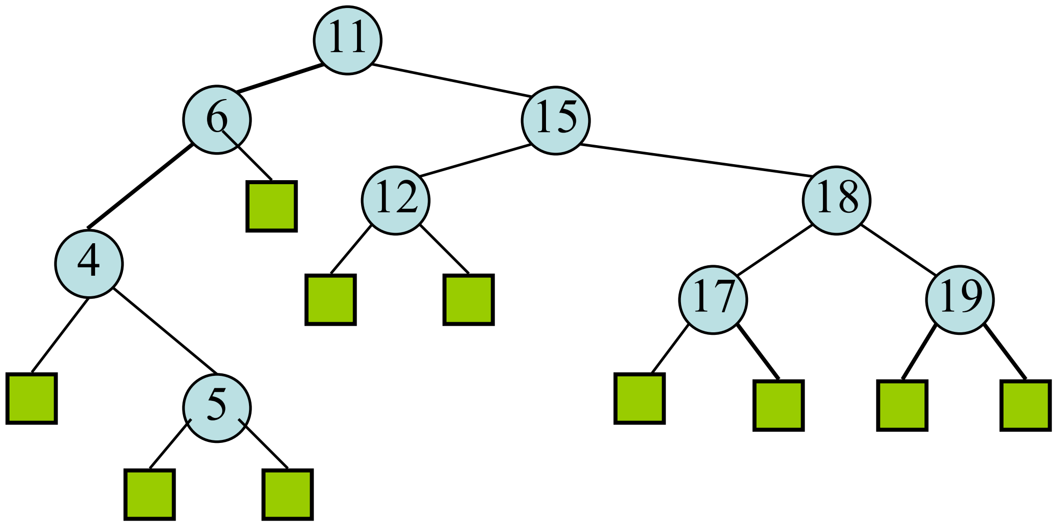
# Delete 13



# Delete 13



# Insert 16



# Add 16