Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054 ADE
Binary Search Trees

Note to help with understanding: "Global vs. Local Perspectives"

(intended to help with coding/understanding algorithms)

- Global view of tree (or any other data structure)
  - can look at entire tree in one go
  - human seeing a picture of a (small) tree
- Local view of tree
  - what your code sees: "code perspective"
  - code generally only sees a local portion of a tree and must work with that
- Vital coding skill: develop ability to see the local view, "code perspective", and not only the global
  - at each code line need to know which data is immediately accessible
  - Avoid trying to code in a way that cannot be done locally
  - Note the local view still needs to work towards a global goal

Note intended to help with your coding: "Global vs. Local Perspectives" Analogies

- Your Data Structure:
  - A real tree
- Your code:
  - "A short-sighted ant crawling over the tree"

Note intended to help with your coding: "Global vs. Local Perspectives" Analogies

- Your Data Structure:
  - A big underground cave system
- Your code:
  - You!
  - … with a weak flashlight
  - … and no overall map
  - … but a goal of which cavern to reach

Motivations

- Suppose you have an array A (of ints) and want to search for a particular int k
- If the array is unsorted then no choice but to scan all elements, hence O(n)
- If it is sorted then we can do a lot better. How?

Motivations

- Binary Search for an element within a sorted array:

```
Algorithm BinarySearch(int[] A, int k, int L, int R)
// searches for k in the range A[L] … A[R] of the sorted array A
// (note the standard technique of working with a sub-array)
    if R < L
        return false
    m = floor( (L + R) / 2  ) // an approximate midpoint
    if k == A[m]
        return true
    else if k < A[m]
        return BinarySearch(A, k, L, m-1)
    else  //  k > A[m]
        return BinarySearch(A, k, m+1, R)
```

Motivations

- Binary Search for an element within a sorted array is fast
  - The array to be searched is halved at each iteration
  - Hence O(log n) (Next slide proves this).
- It only works because of the step of
  "knowing whether to go left or right"
- But arrays suffer from being slow, O(n), to insert new elements
  - because need to shift O(n) elements to make room for them

- **Search trees attempt to fix the inefficiency of insertion whilst keeping good O(log n) properties of binary search**
  - **So, we need a tree where we "know which way to go":**

Binary search complexity

- Using recurrence relations
  - T(n) = T(n/2) + 1
    - "n/2" because only have to search ½ the array after recursion
    - "1" as the cost of doing the comparison and deciding whether to go left or right
  - Can do exactly (exercise!)

- Use Master Theorem (revise lectures if needed):
  - a=1, b=2, log_b(a) = log_2(1) = 0
  - So c = log_b(a) = 0
  - Have f(n) = 1 which is Θ(n^0)
  - So is Case 2: and then is n^0 but "get an extra log"
  - Hence, Θ(log(n))

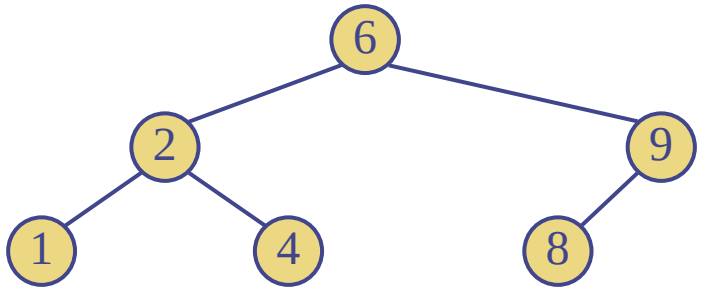
** Binary Search Trees **

- A binary search tree is a binary tree storing key-value entries at its internal nodes and satisfying the following "search tree" property:
  - Let u, v, and w be any three nodes such that u is in the **left subtree** of v and w is in the **right subtree** of v. Then we must have

    $$key(u) \leq key(v) \leq key(w)$$

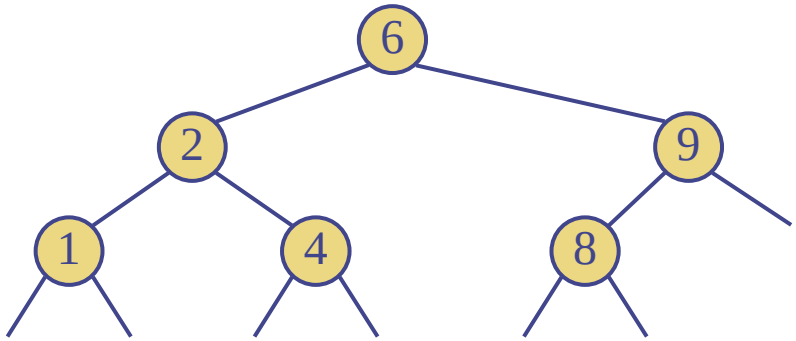
    or, as we will assume there are no duplicate keys:

    $$key(u) < key(v) < key(w)$$



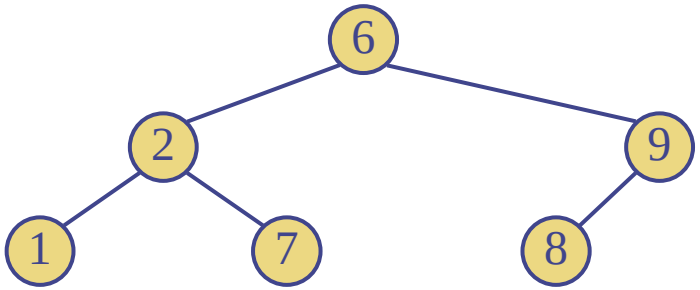
Binary Search Trees

- “”Null” children
  - Sometimes, just for for clarity, we explicitly show the “null pointers” that are for the case of a non-existent child



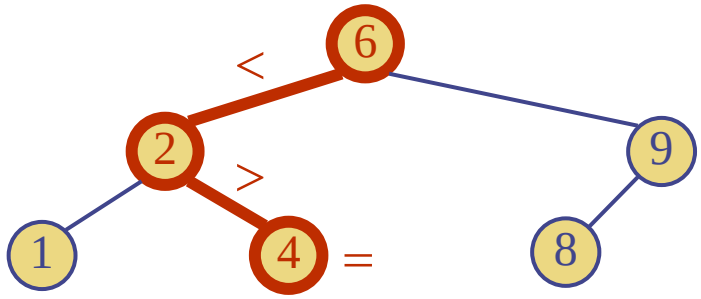
Not a Binary Search Tree

- VERY IMPORTANT:
- The following is **not** a Binary Search Tree
  - Even though the immediate children of every node look correct, we need that **every** node in the left subtree of 6 is less than 6, and this is failed by the node '7'



Search

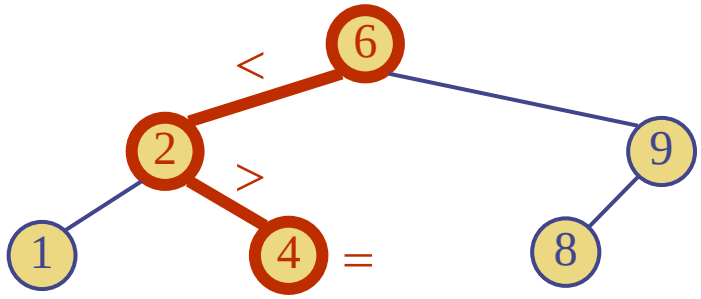
- To search for key k, trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return null



Search

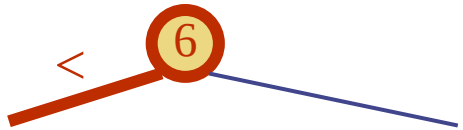
```
Algorithm Node TreeSearch(Key k, Node n)
    if n.isExternal () // or, "if n == null"
        return null
    if k < n.key()
        return TreeSearch(k, n.left())
    else if k = n.key()
        return n
    else // k > n.key()
        return TreeSearch(k, n.right())
```

- Example: find(4):
  - Call TreeSearch(4,root)
  - Note: standard trick of using an auxiliary function that keeps more of the "state"
- Compare this with the binary search of an array



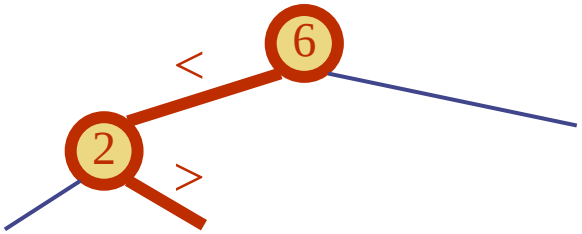
Search from "code perspective"

- Example: find(4):
- 4 < 6 so need to go left



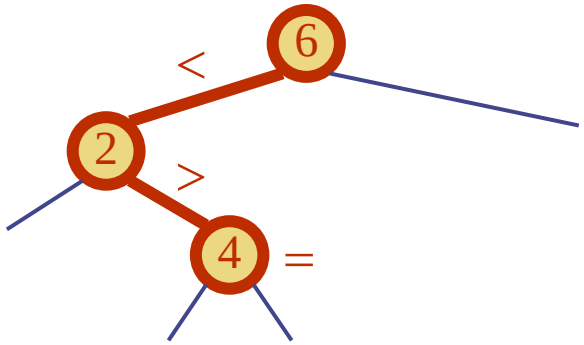
Search from "code perspective"

- Example: find(4):
- 4 > 2 so need to go right



Search from "code perspective"

- Example: find(4):
- Found 4, but note that we only saw/visited a small part of the entire tree



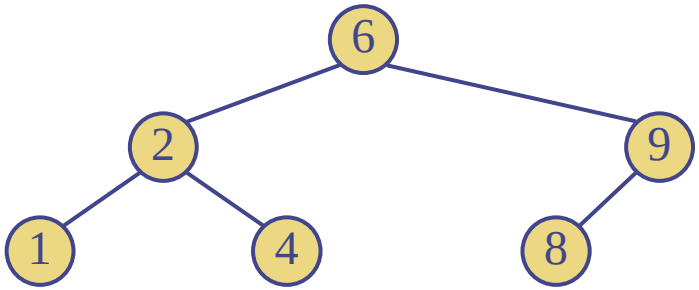
- It only follows a path from the root to some node, and so is clearly O(h) where h is the height of the tree.

Recursive vs. Iterative

- Generally: Recursive programs are easier to implement, but less efficient
  - because of the overhead of a function call
- For best efficiency, will need to convert to an iterative program
  "while (test) { … }"
- Exercises (offline): convert to iterative
  - binary search of an array
  - search in a binary search tree

Fundamental Property of Search Tree

- Exercise: what does an inorder traversal of the following search tree produce?



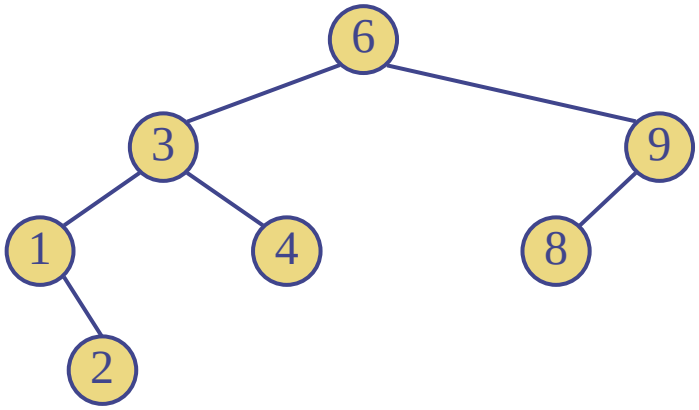
Fundamental Property of Search Tree

Directly from the definition:

- An inorder traversal of a (binary) search trees visits the keys in increasing order

Fundamental Property of Search Tree

- Note that to access the minimum key, we just need to 'always go left'



Insertion

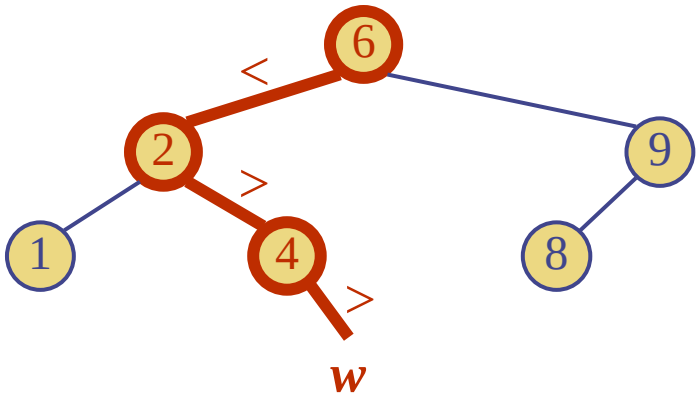
- Exercise: how to do insertion, insert(k, v) ?

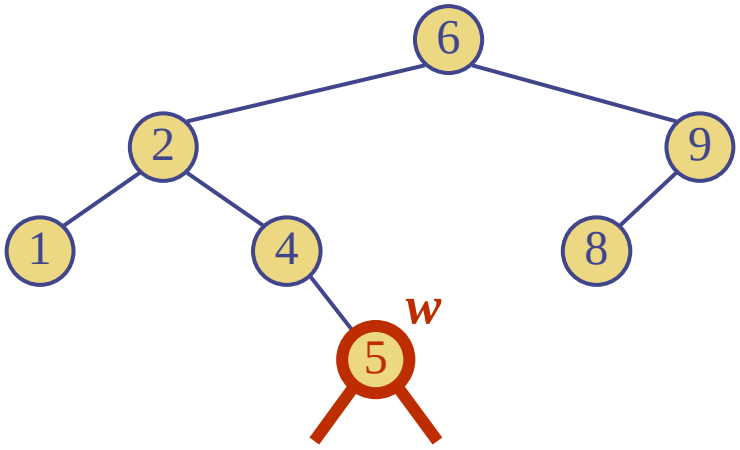
- Have to insert k where a get(k) would find it!
- So natural that insert(k,v) starts with get(k)

- We search for key k (using TreeSearch)
- If k is already in the tree then just replace the value
- Otherwise, k is not already in the tree, and let w be the leaf reached by the search
  - We "insert k at node w and expand w into an internal node"
- Again, only follows a path from the root and so is O(h)

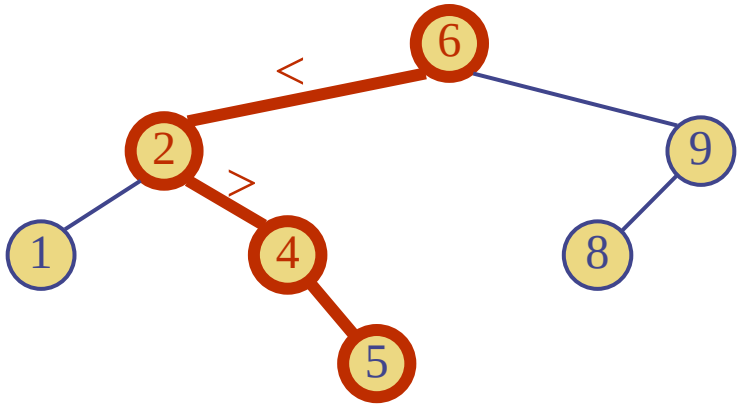
Insertion

- Example: insert 5
- We search for key k=5 (using TreeSearch)
- Let w be "missing child position" reached by the search
- We insert a real node, with key k, at that location w
- Exercise: check that this is correct! Does it preserve the search tree property?





Deletion

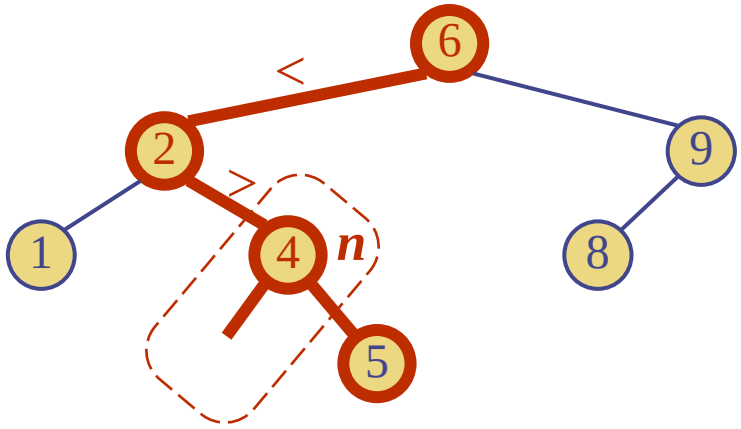


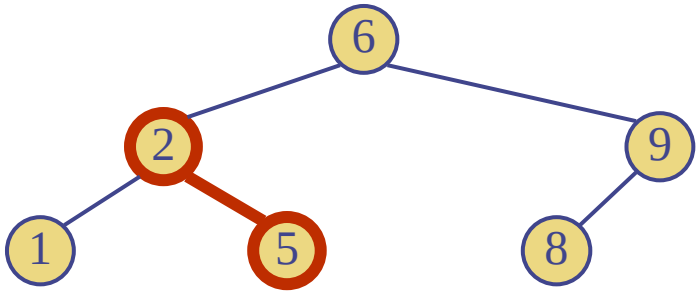
- How can we perform operation remove(k) ?
- As usual we start by trying to find(k)
- Four cases: (think of the externals as null – not as real children)

1. k is not present, e.g. remove(7)
   nothing to do
   otherwise k is stored in some node n
2. n has no children, e.g. remove(5),
   (straightforward, left as an exercise)
3. n has one child, e.g. remove(4)
4. n has two children, e.g. remove(2)

Deletion – with one child

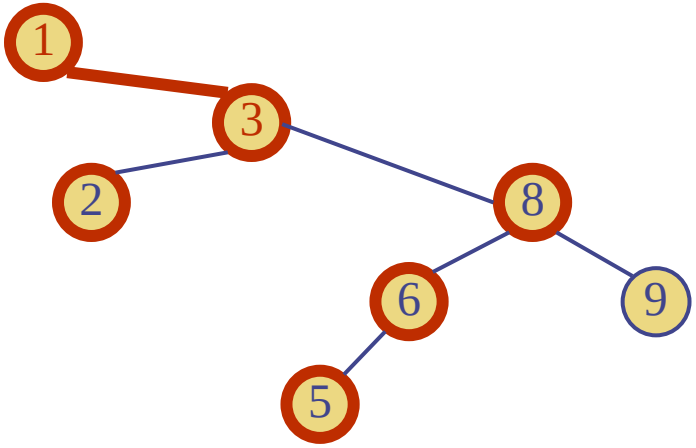
- Example: remove 4
- To perform operation remove(*4*), we search for key *4*.
- Let ***n*** be the node storing *4*.
- Node ***n*** has a null left child, and a real child 5
- We remove ***n*** from the tree and connect 5 back to the parent of ***n***
- **Note that this still works if n is a right child and has a left child. (Exercise).**





Deletion – with two children

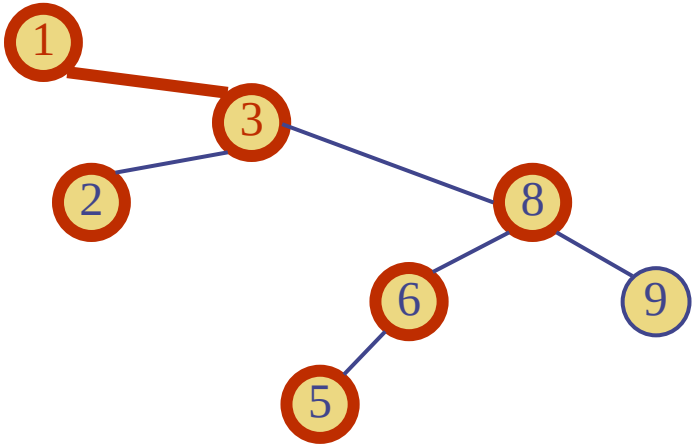
- Example: remove 3
- **What can we do and still keep the property of being a search tree?**



- "Search tree" means an "inorder traversal" visits the keys "in order"
- As a sorted list, we would have [1,2,3,5,6,8,9]
- If we want to remove '3' then a way to do this with minimal change to the rest of the list is to copy a key k' that is adjacent to '3' on top of '3' and then delete that key k'
    - Options in this case are k' being 2 or 5. We will focus on the nextKey, that is, '5'

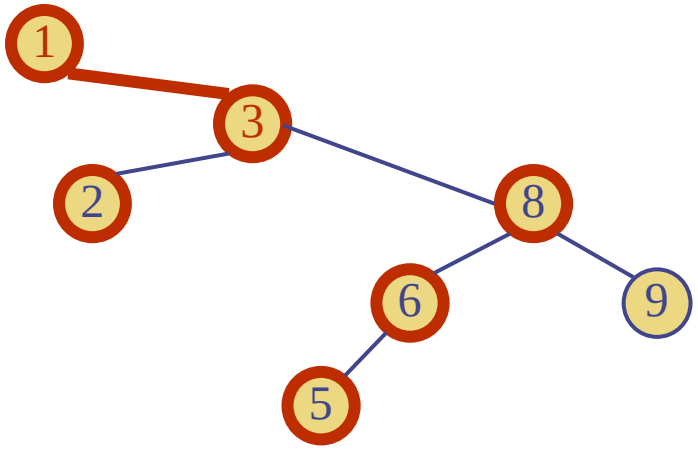
Deletion – with two children

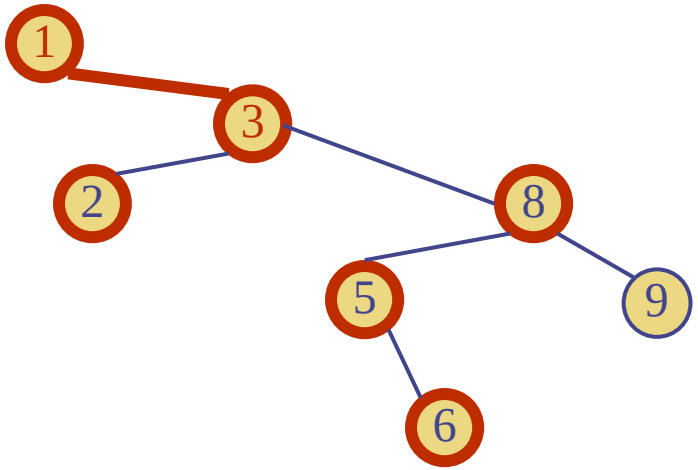
- Example: remove 3
- **What can we do and still keep the property of being a search tree?**



- "Search tree" means an "inorder traversal" visits the keys "in order"
- If these were a sorted list then "adjacent in the sequence" nodes 2 and 5 could be involved in the removal of node 3
- Can the next key after 3 ever have any left children?
- Can the next key after 3 ever have any children?

Deletion – with two children





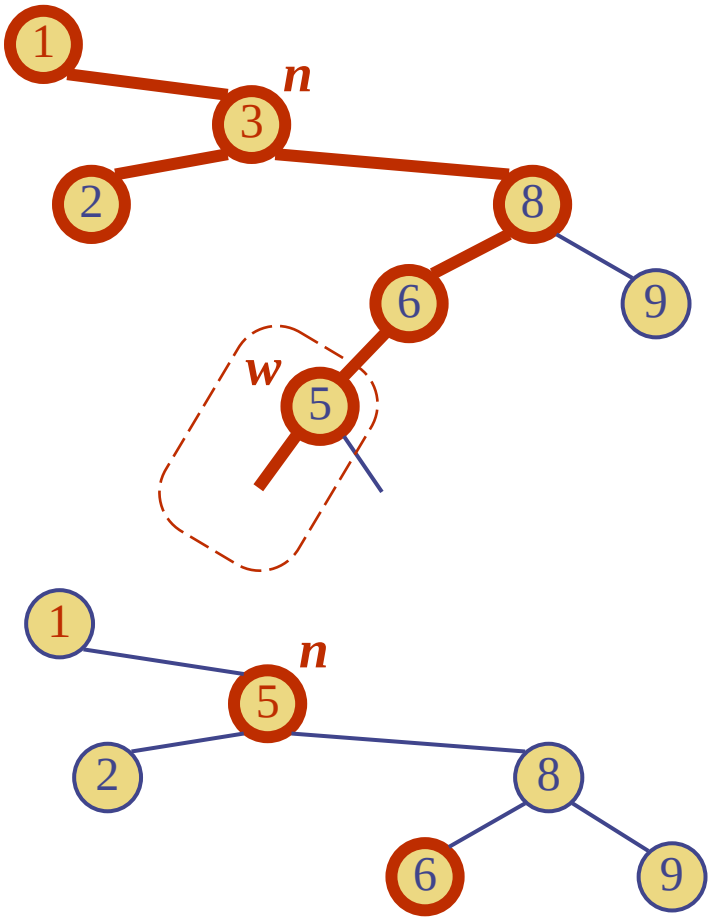
- Can the next (using inorder) key after 3 ever have any children?
- YES. The right hand tree is also a possible search tree with same keys
- To access the nextKey, we need to go right, and then recursively-go-left
- IMPORTANT: that that the next key from 3 cannot have a left child: Proof by contradiction. If it did then the left child would not be "nextKey"

Deletion – with two children

Example: remove 3

The key node n has two internal children

1. we find the internal node w that follows n in an inorder traversal
2. we copy $key(w)$ into node n
3. we remove node w by means of same procedure as before for "one child"



Deletion – with two children

General procedure:

- Consider the case where the key k to be removed is stored at a node n whose children are both internal
  - we find the internal node w that follows n in an inorder traversal
  - we copy $key(w)$ into node n
  - we remove node w and its left child z (which must be a leaf) by using the same procedure as discussed earlier for "remove node with one child"

- Exercises (offline):
  - convince yourself this works properly even when w has a right child
  - Check that it is still O(h)

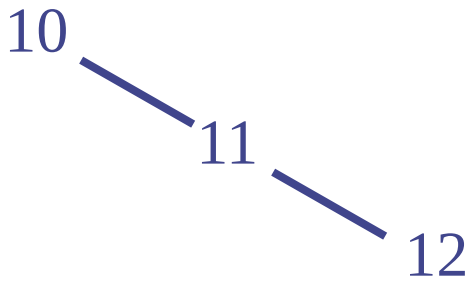
Exercises (offline)

In the "2 children deletion", we had "we find the internal node that follows n in an inorder traversal"

1. Could it also have been "precedes" instead of "follows"?
2. Might making a good choice of "follows" of "precedes" have some advantages?

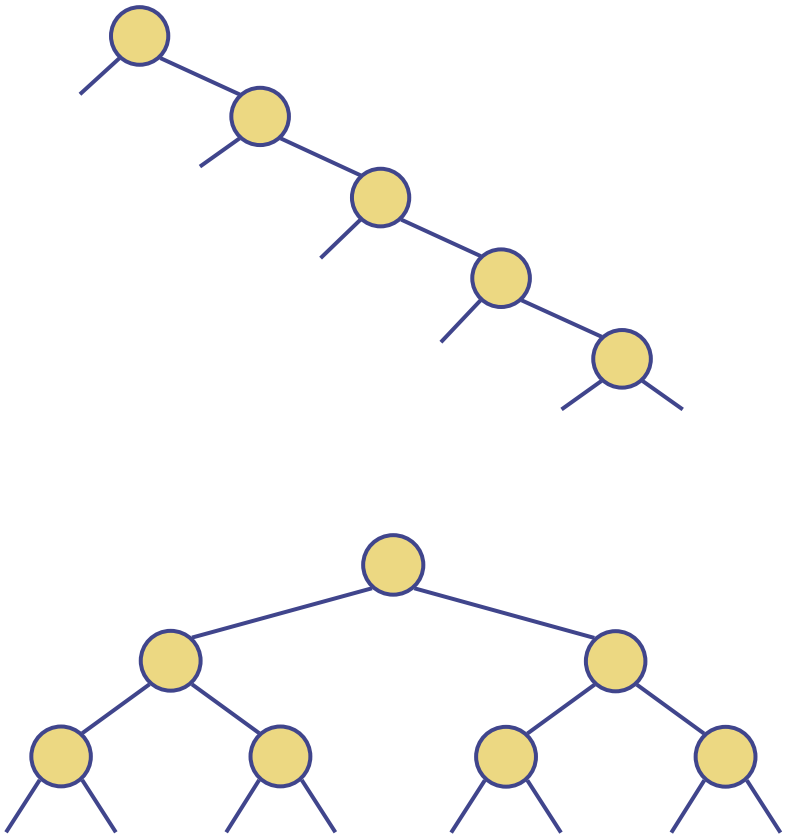
Balanced Trees

- Binary search trees: if all levels filled, then search, insertion and deletion are O(log N).
  - As they are all O(height)
- However, performance may deteriorate to linear if nodes are inserted in order:



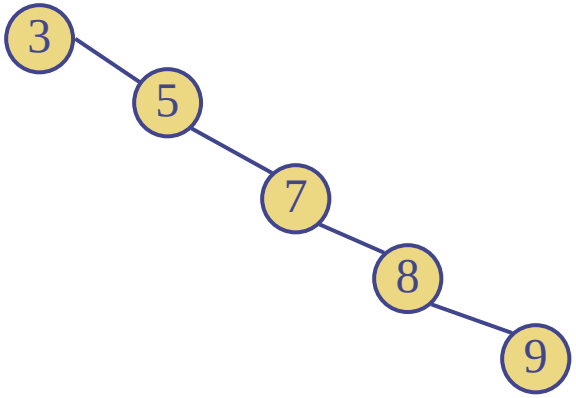
Performance

- Consider a binary search tree of height h with n items
  - the space used is $O(n)$
  - methods find, insert and remove take $O(h)$ time
  - EXERCISE: carefully study the methods and check that this is true
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case

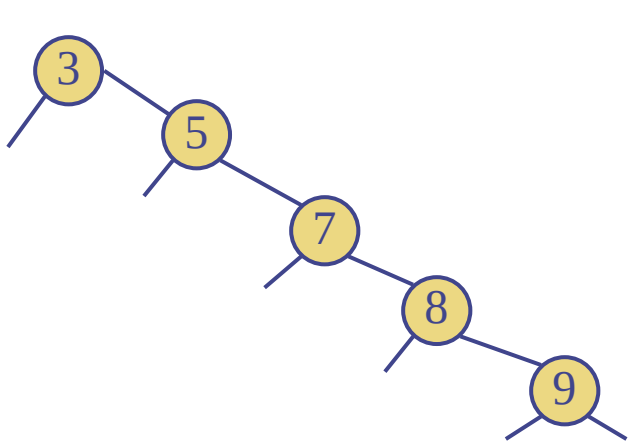


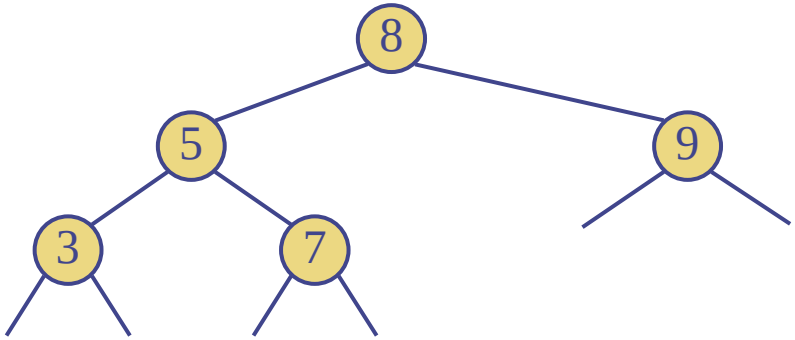
Issues in "Self-Balancing"

- Suppose you have a very imbalanced search tree – is there always a corresponding balanced search tree?
- Exercise (online): find a balanced tree for



Example





- For any number of nodes, there is (obviously?) a balanced tree
- Could balance by a "total rebuild"
  - just placing the keys into a new tree in the correct order

Solution: Self-Balancing

Goal of "Self-Balancing":

- Constantly re-structure the trees:
- Keep the trees height balanced so that the height is logarithmic in the size
- Performance always logarithmic.

Issues in Self-Balancing

- Suppose you have a very imbalanced search tree, there are always corresponding balanced search trees
- Could make trees balanced using a "total rebuild"
  - But would require O(n), and so very inefficient compared to the desired O(log n)
- Re-balancing needs to be O(log n) or O(height)
- Suggests re-balancing needs to just look at the path to some recently changed node, not the entire tree
- A priori, it is not at all obvious that this is possible!
- In a later lecture we show how to use "rotations" to adjust the tree whilst preserving the ordering

Minimal Expectations

- The definitions of Map ADT, BST, etc
- How to use binary search trees
- The complexity of operations as function of nodes *n* and height *h*