Binary Search Trees

cse2011

section 10.1 of textbook

Dictionary ADT (section 9.5.1)

- The **dictionary** ADT models a searchable collection of key-element items
- The **main operations** of a dictionary are **searching**, **inserting**, and **deleting** items
- **Multiple items** with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - SIN database
  - student database

Dictionary ADT methods:

- get(k): if the dictionary has an item with key k, returns its element, else, returns NULL
- getAll(k): returns an iterator of entries with key k
- put(k, o): inserts item (k, o) into the dictionary, which k is the key and o is the object
- remove(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns NULL
- removeAll(k): remove all entries with key k; return an iterator of these entries.
- size(), isEmpty()

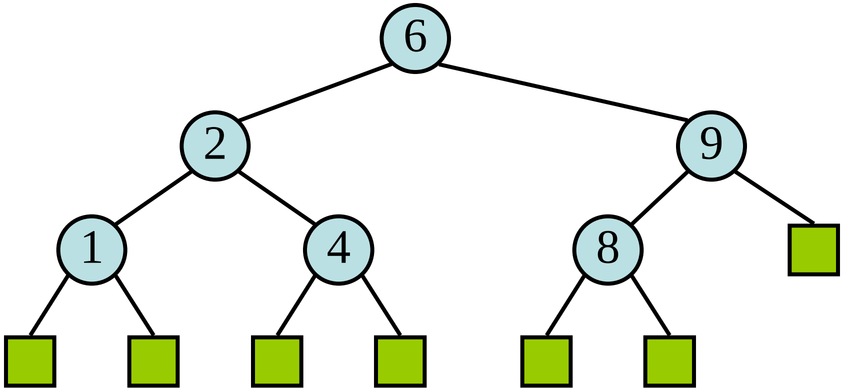
Binary Search Trees

- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property:

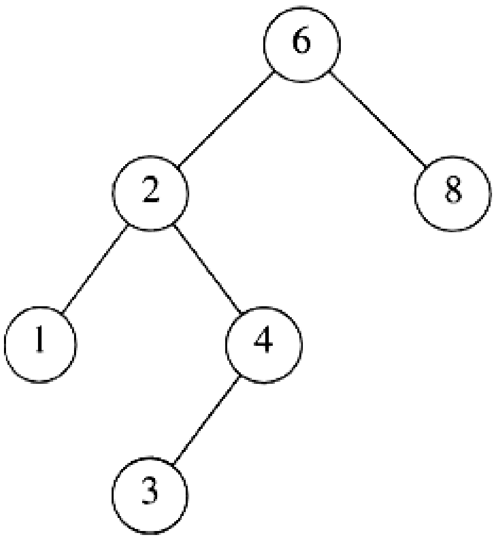
  Let ***u***, ***v***, and ***w*** be three nodes such that ***u*** is in the left subtree of ***v*** and ***w*** is in the right subtree of ***v***. We have
  $key(u) \leq key(v) \leq key(w)$

- External nodes (dummies) **do not** store items (non-empty proper binary trees, for coding simplicity)

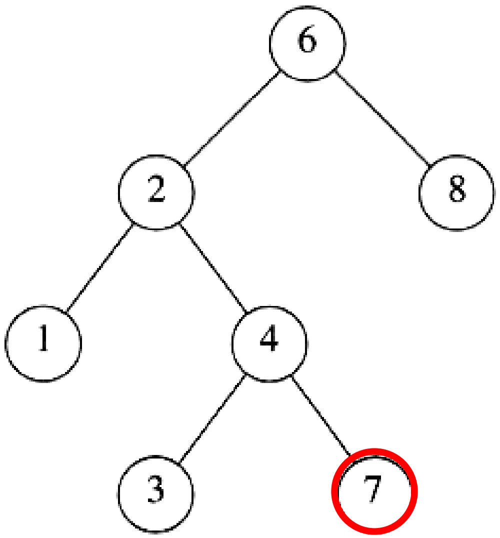
- An **inorder** traversal of a binary search trees visits the keys in increasing order
- The left-most child has the **smallest** key
- The right-most child has the **largest** key



Example of BST



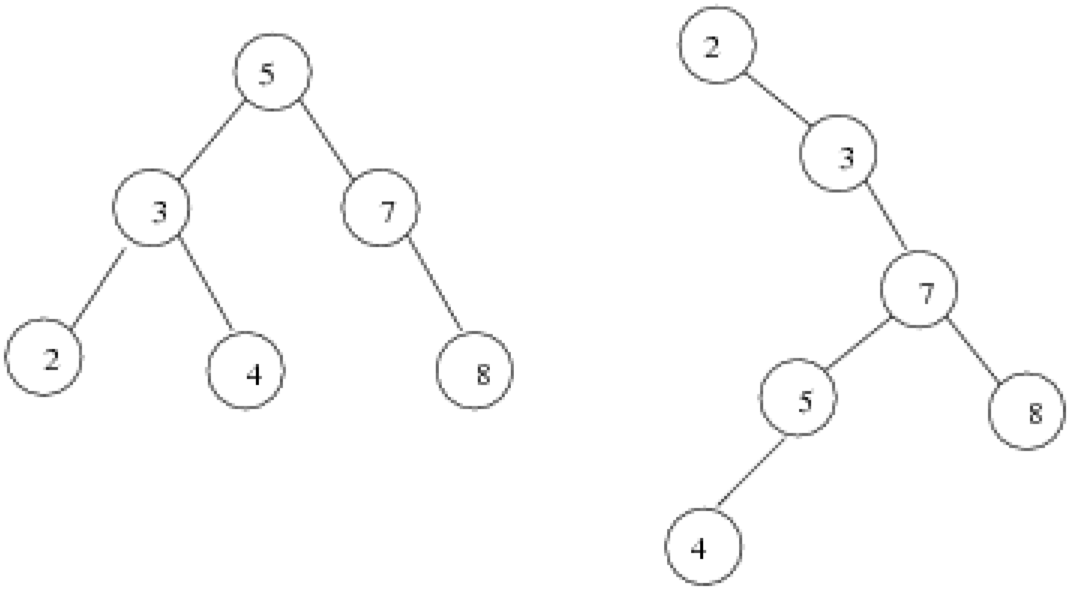
A binary search tree



Not a binary search tree

More Examples of BST

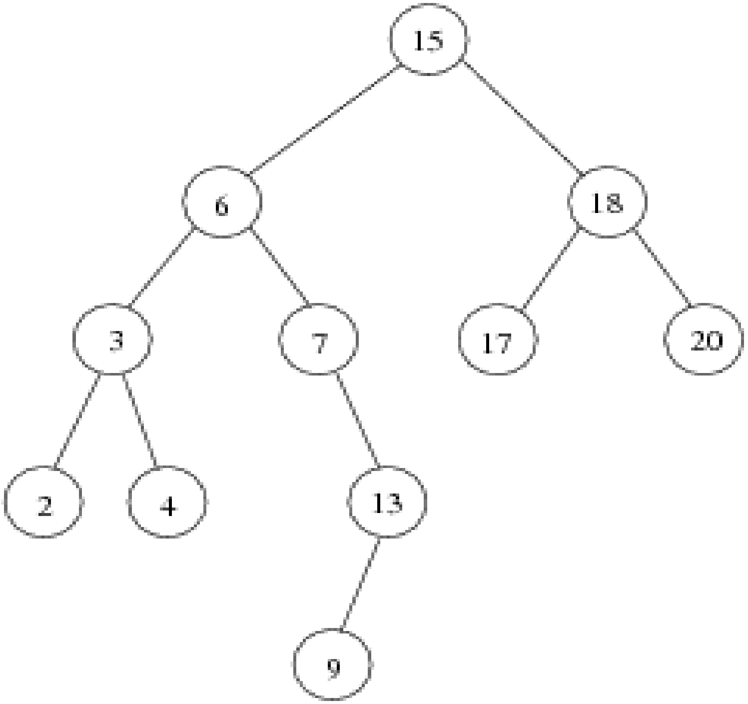
The same set of keys may have different BSTs.



- Average depth of a node is O(logN).
- Maximum depth of a node is O(N).
- Where is the smallest key? largest key?

Inorder Traversal of BST

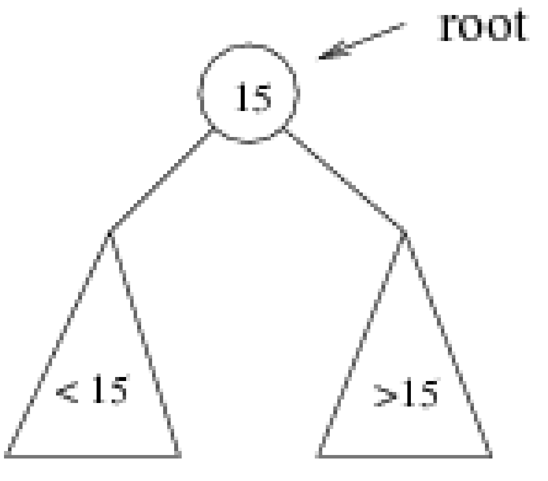
- Inorder traversal of BST prints out all the keys in sorted order.



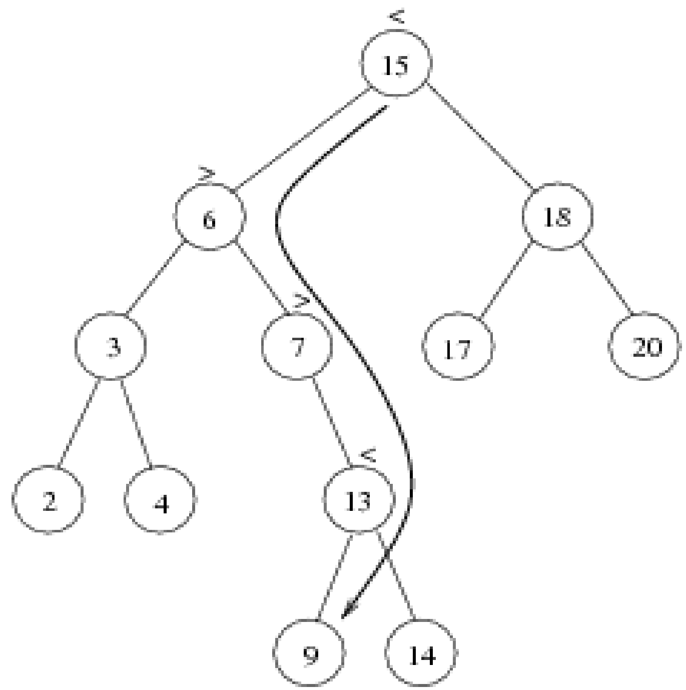
Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Example: Search for 9 ...



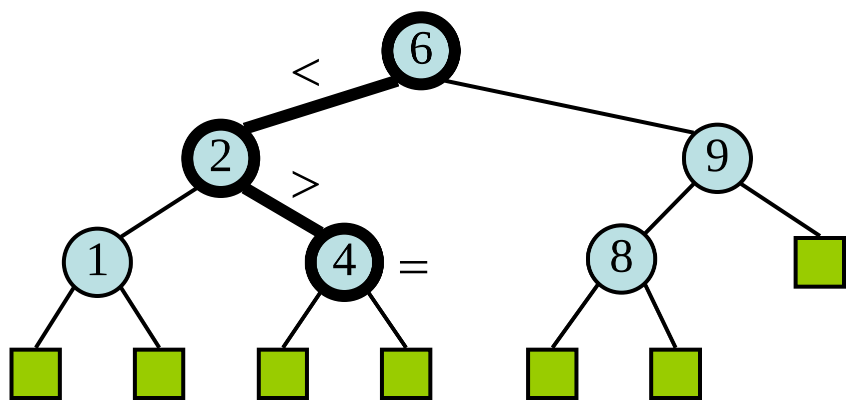
Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Search Algorithm

- 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 v (where the key should be if it will be inserted)
- Example: TreeSearch(4, T.root())
- Running time: ?

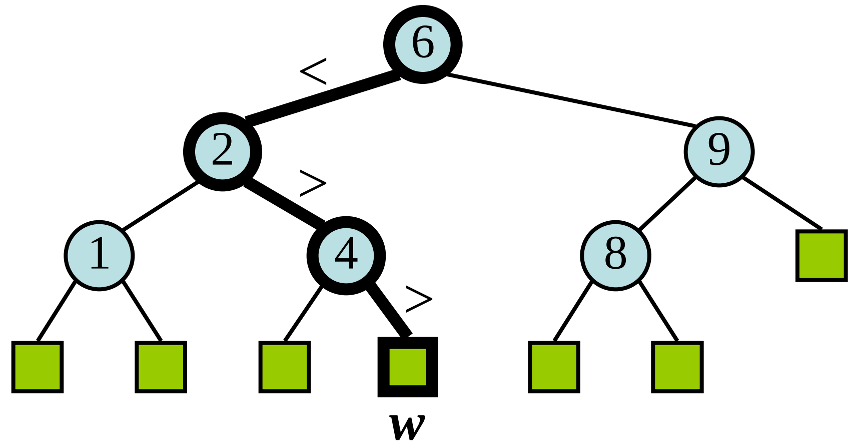
```
Algorithm TreeSearch( k, v )
    if T.isExternal (v)
        return (v);     // or return NO_SUCH_KEY
    if k < key(v)
        return TreeSearch( k, T.left(v) )
    else if k = key(v)
        return v
    else //{ k > key(v) }
        return TreeSearch( k, T.right(v) )
```

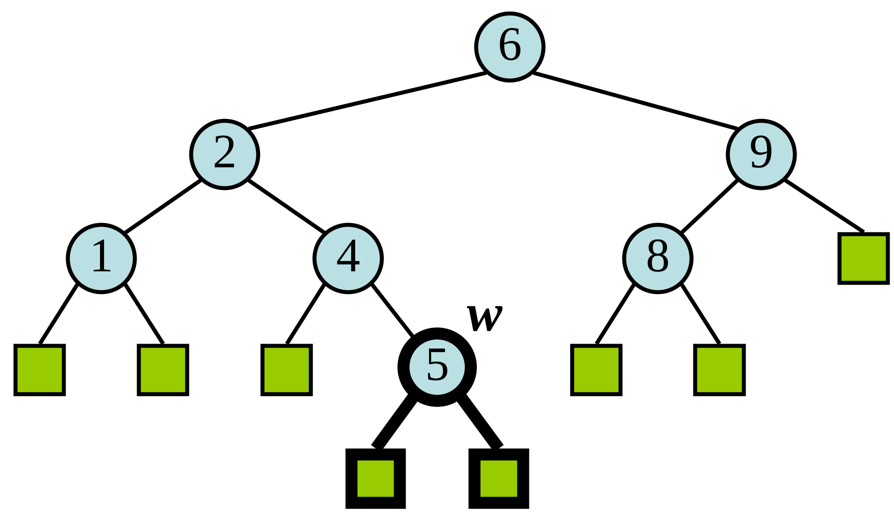


Insertion (distinct keys)

- To perform operation insertItem(k, o), we search for key k
- Assume 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 using *insertAtExternal*(w, (k,e))
- Example:

  insertAtExternal(w, (5,e)) with e having key 5
- Running time: ?





Insertion Algorithm (distinct keys)

```
Algorithm TreeInsert( k, e, v ) {
    w = TreeSearch( k, v );
    T.insertAtExternal( w, k, e );
    return w;
}
Algorithm insertAtExternal( w, k, e ) {
   if ( T.isExternal( w ) {
      make w an internal node, store k and e into w;
      add two dummy nodes (leaves) as w's children;
   }  else  { error condition };
}
```

- First call: TreeInsert(5, e, T.root( ))

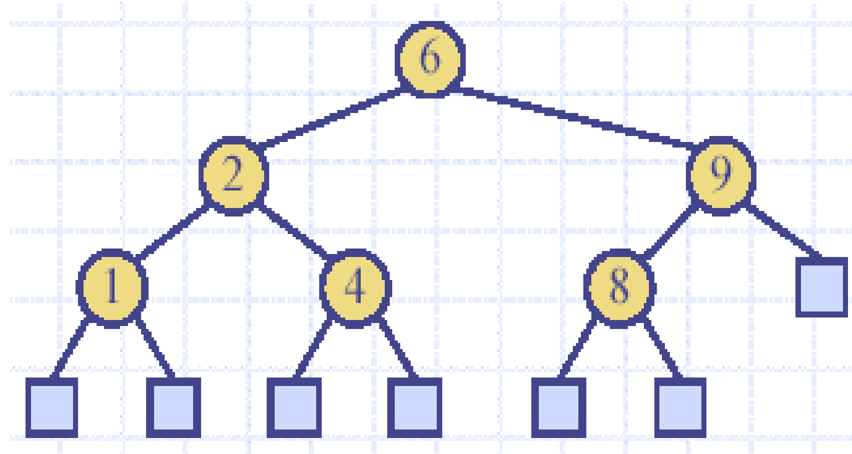
Insertion (duplicate keys)

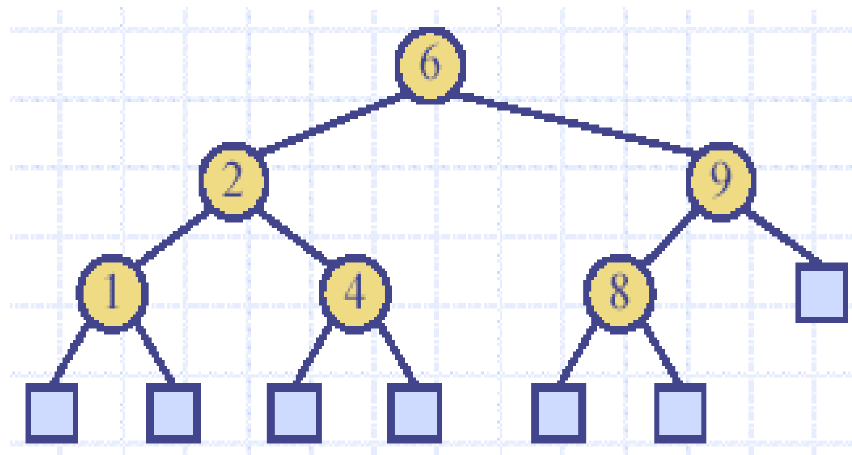
Insertion with duplicate keys

- Example: insert(2)
- Call *TreeSearch*(k, *leftChild*(w)) to find the leaf node for insertion
- Can insert to either the left subtree or the right subtree (call *TreeSearch*(k, *rightChild*(w))

Running time: ?

Homework: implement method getAll(k)





Insertion Algorithm (duplicate keys)

```
Algorithm TreeInsert( k, e, v ) {
    w = TreeSearch( k, v );
    if k == key(w)   //  key exists
       return TreeInsert( k, e, T.left( w ) );     // ***
    T.insertAtExternal( w, k, e );
    return w;
}
```

- First call: TreeInsert(2, e, T.root())

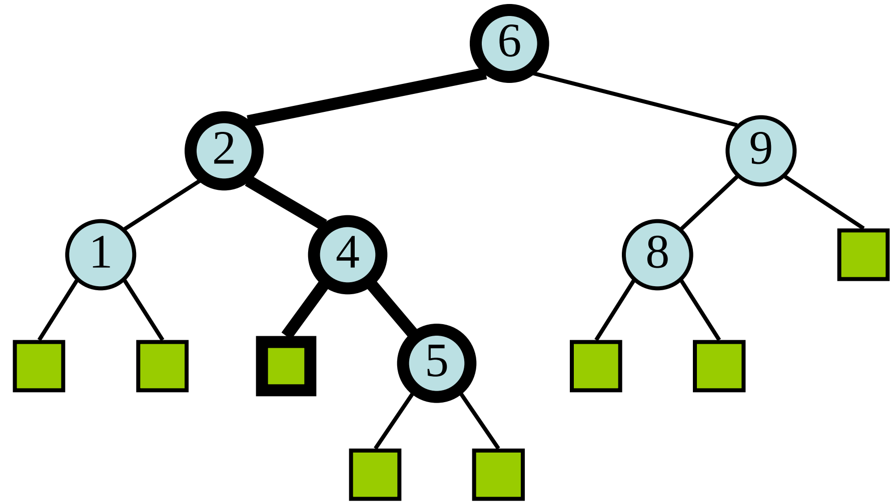
Note: if inserting the duplicate key into the ***left subtree, keep searching the ***left*** subtree after a key has been found.

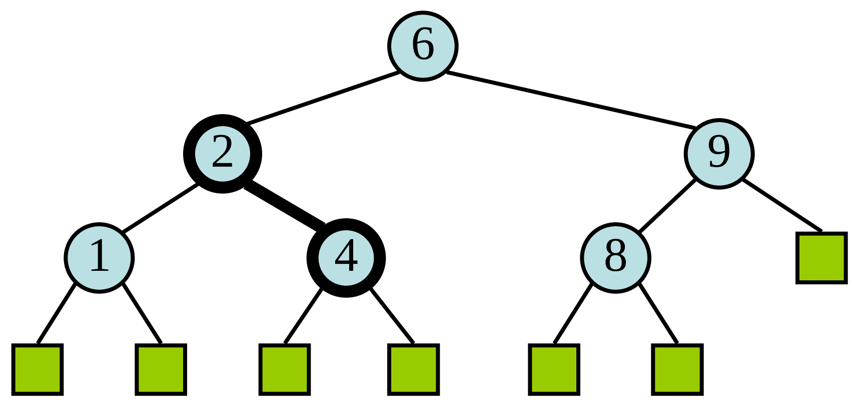
Deletion

- To perform operation removeElement(***k***), we search for key ***k***
- Assume key ***k*** is in the tree, and let ***v*** be the node storing ***k***
- **Three** cases:
  - Case 1: ***v*** has no children
  - Case 2: ***v*** has exactly one child
  - Case 3: ***v*** has two children

Deletion: Case 1

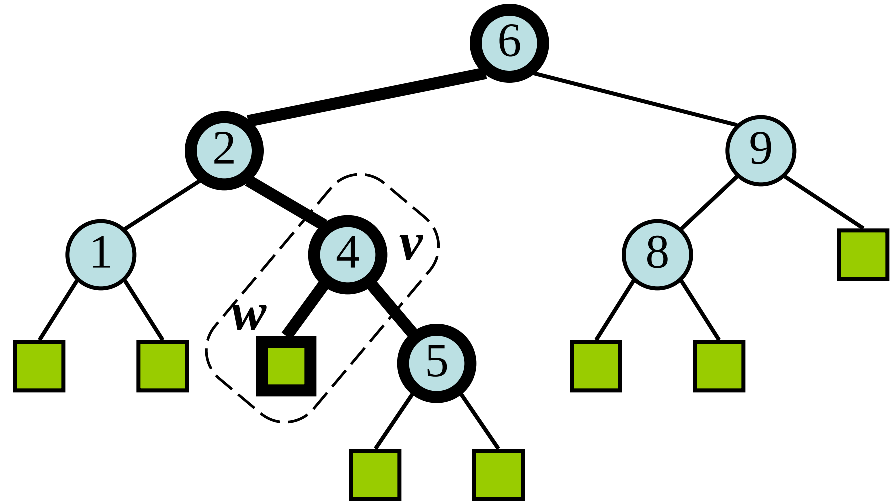
- Case 1: ***v*** has no children
- We simply remove ***v*** and its 2 dummy leaves.
- Replace ***v*** by a dummy node.
- Example: remove 5

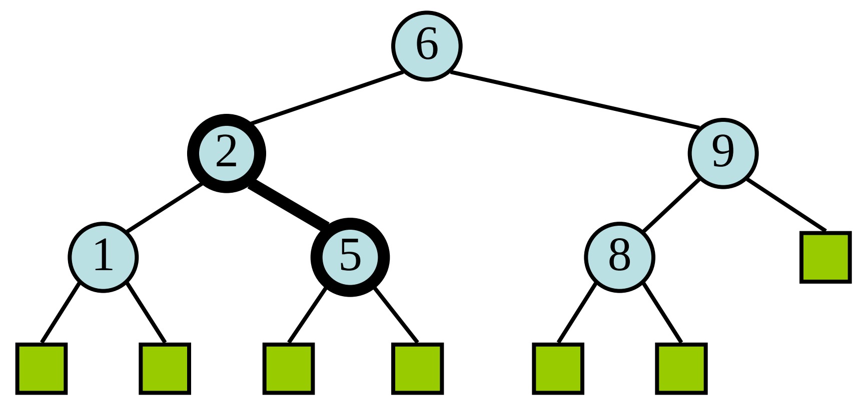




Deletion: Case 2

- Case 1: v has exactly one child
- v's parent will "adopt" v's child.
- We connect v's parent to v's child, effectively removing v and the dummy node w from the tree.
- Done by method **`removeExternal`**(w)
- Example: remove 4





Method *removeExternal*( )

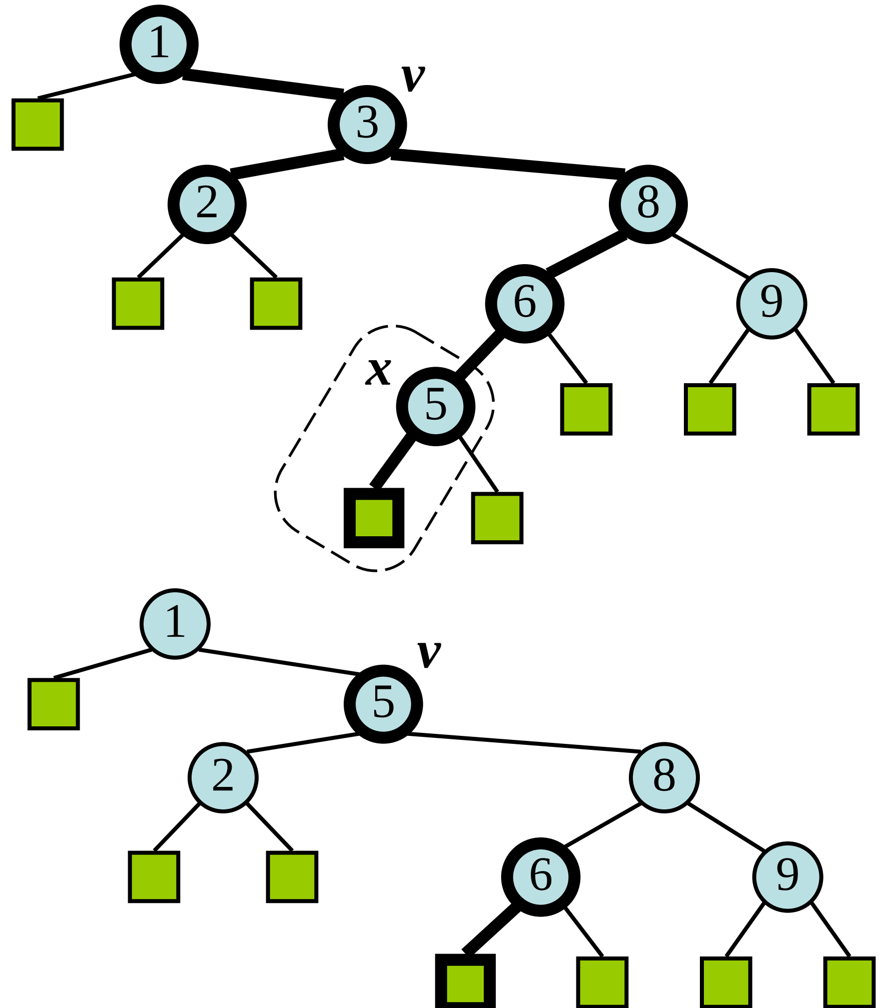
- Remove an external node v and its parent:
  - Replacing v's parent with v's sibling
  - An error occurs if v is not external

Deletion: Case 3

- Case 3: ***v*** has **two** children (and possibly grandchildren, great-grandchildren, etc.)
- Identify ***v***'s "heir": *either* one of the following two nodes:
  - the node ***x*** that immediately precedes ***v*** in an **inorder** traversal (**right-most** node in ***v***'s left subtree)
  - the node ***x*** that immediately follows ***v*** in an **inorder** traversal (**left-most** node in ***v***'s right subtree)
- Two steps:
  - **copy** content of ***x*** into node ***v*** (heir "inherits" node ***v***);
  - **remove** ***x*** from the tree (use either case 1 or case 2 above).

Deletion: Case 3 Example

- Example: remove 3
- Heir = ?



- Running time of deletion algorithm: ?
- Homework: implement removeAll(k)

Notes

- Two steps of case 3:
  - copy content of ***x*** into node ***v*** (heir "inherits" node ***v***);
  - remove ***x*** from the tree
    - if ***x*** has no child: call case 1
    - if ***x*** has one child: call case 2
    - ***x*** cannot have two children (why?/Homework)

- Both cases 1 and 2 can be merged into one and implemented by method **`removeExternal()`**.

Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods get(k) , put() and remove(k) take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case