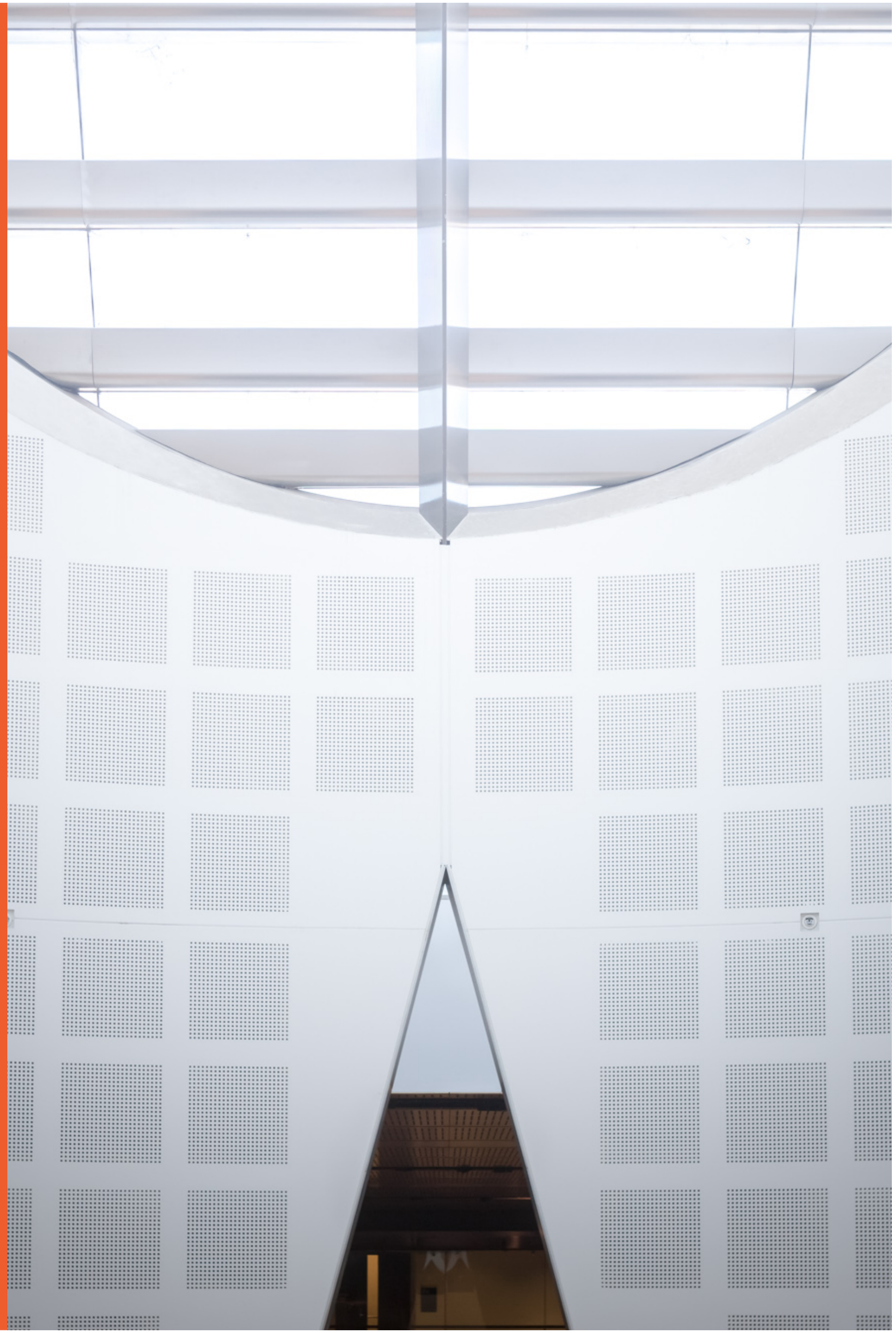
INFO1105/1905
Data Structures
Week 6: Map, Binary Search Tree

see textbook section 10.1, 10.3, 11.1

Professor Alan Fekete
Dr John Stavrakakis
School of Information Technologies

using material from the textbook
and A/Prof Kalina Yacef

THE UNIVERSITY OF SYDNEY



Copyright warning

- These slides contain material from the textbook (Goodrich, Tamassia & Goldwasser)
  - Data structures and algorithms in Java (5^{th} & 6^{th} edition)
- With modifications and additions from the University of Sydney

- The slides are a guide or overview of some big ideas
  - Students are responsible for knowing what is in the referenced sections of the textbook, not just what is in the slides

Reminder! Asst 1

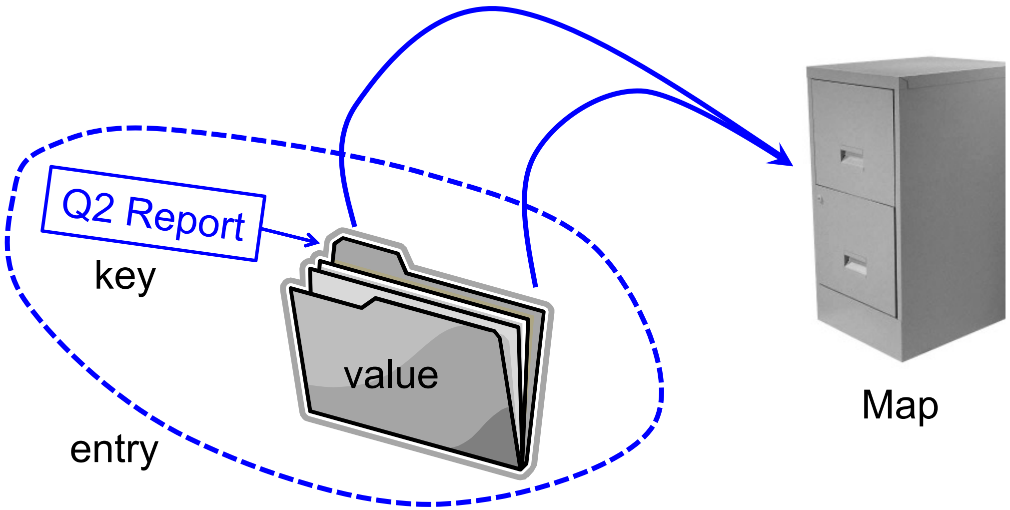
- Asst 1 will be released by Friday this week (Sept 2)
- Due 5pm Friday Sept 16
- Develop an application that *uses* appropriate collection types to deliver required functionality, efficiently
- Two aspects to submit:
  - Report (in pdf, not hand-written), submit via Turnitin link on eLearning site
  - Code (including Junit tests), submit via edstem
- Individual work. All use of ideas and material from other people must be properly acknowledged (and quoted, if using their words or code)
- Marking will be based on automarking (public and private tests), handmarking (design, style, etc), and the report

Outline

- Map ADT (section 10.1)
- Sorted Map ADT (section 10.3)
- Binary Search Trees (section 11.1)
  - Definition
  - Searching
  - Operations on BSTs
  - Performance

Maps

- A map models a searchable collection of key-value entries
  - Elements can be located quickly using keys
- Key = unique identifier
  - Multiple entries with the same key are not allowed
- The main operations of a map are for
  - searching,
  - inserting, and
  - deleting items
- Applications:
  - address book
  - student-record database
  - Web



The Map ADT

- get(k): if the map M has an entry with key k, return its associated value; else, return **null**
- put(k, v): ; if key k is not already in M, then insert entry (k, v) into the map M and return null; else, replace the existing value associated to k with v, and return the value previously associated to k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return **null**
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterable collection of the values in M

Example

Operation	*Output*	*Map*
isEmpty()	**true**	∅
put(5,*A*)	**null**	(5,*A*)
put(7,*B*)	**null**	(5,*A*),(7,*B*)
put(2,*C*)	**null**	(5,*A*),(7,*B*),(2,*C*)
put(8,*D*)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
put(2,*E*)	*C*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(7)	*B*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(4)	**null**	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
get(2)	*E*	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
size()	4	(5,*A*),(7,*B*),(2,*E*),(8,*D*)
remove(5)	*A*	(7,*B*),(2,*E*),(8,*D*)
remove(2)	*E*	(7,*B*),(8,*D*)
get(2)	**null**	(7,*B*),(8,*D*)
isEmpty()	**false**	(7,*B*),(8,*D*)

When a key is not present

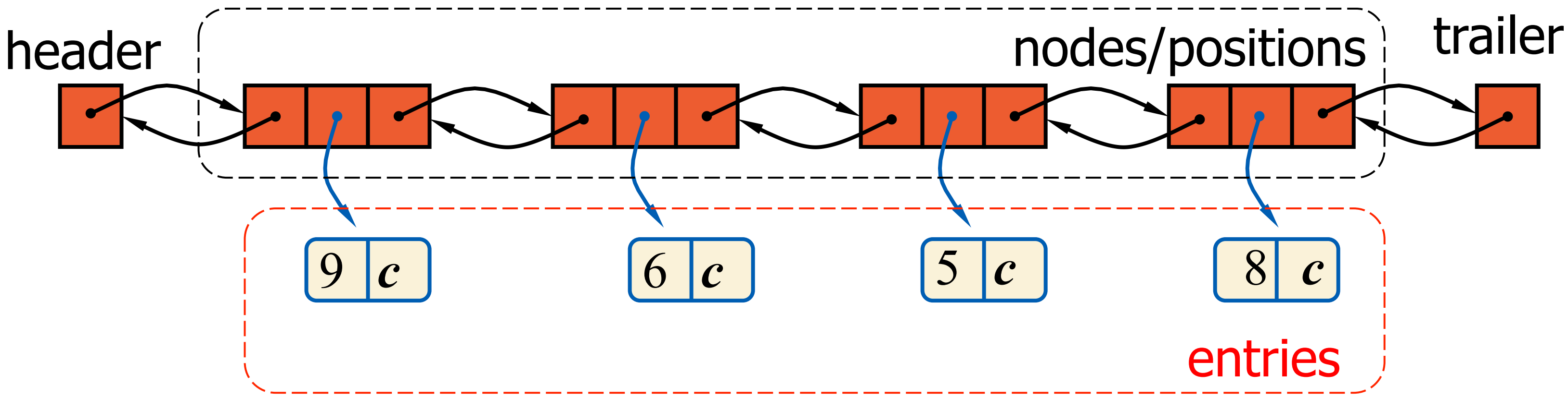
- When operations get(k), put(k,v) and remove (k) are performed on a map that has no entry with key equal to k:
  - Return **null** by convention
  - null is used as a sentinel

- This approach has a disadvantage if null is allowed as a value
  - if an actual entry (k,null) exists, we would also return null for that!

  - So if null might be used as a value, client can check first to distinguish the cases
    - containsKey(k) checks if k exists as a key
  - Alternative API design would raise exceptions, but this makes for messy coding especially involving put

A Simple List-Based (unsorted) Map

- We can implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



The get(k) Algorithm

Algorithm get(k):

B = S.positions() *{B is an iterator of the positions in S}*

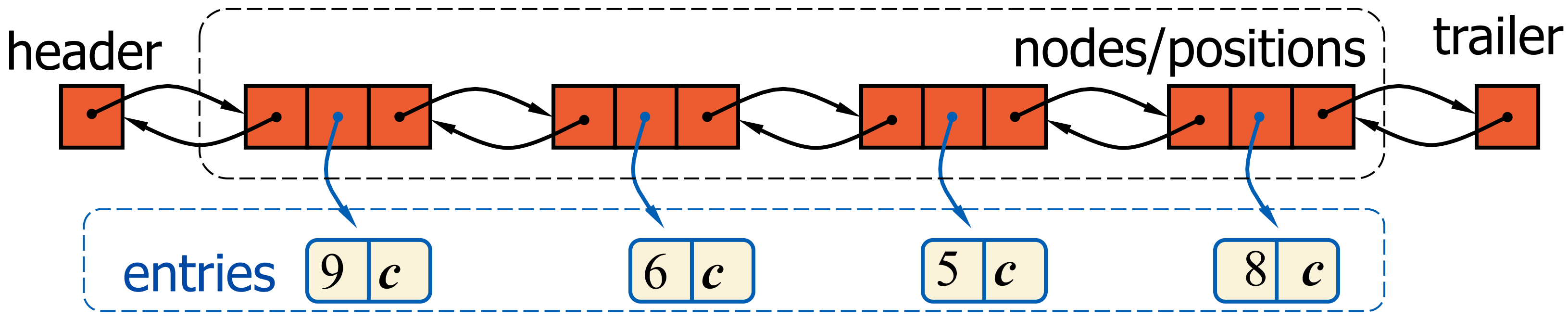
while B.hasNext() **do**

p = B.next() *{ the next position in B }*

if p.element().getKey() = k **then**

return p.element().getValue()

return null *{there is no entry with key equal to k}*



The put(k,v) Algorithm

```
Algorithm put(k,v):
B  = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
     t = p.element().getValue()
     S.set(p,(k,v))
     return t{return the old value}
S.addLast((k,v))
n = n + 1   {increment variable storing number of entries}
return null { there was no entry with key equal to k }
```

The remove(k) Algorithm

```
Algorithm remove(k):
B =S.positions()
while B.hasNext() do
    p = B.next()
    if p.element().getKey() = k  then
     t = p.element().getValue()
     S.remove(p)
     n = n – 1    {decrement number of entries}
     return t{return the removed value}
return null      {there is no entry with key equal to k}
```

Performance of a List-Based Map

- Performance:
  - put takes O(n) time in worst-case as we traverse the list looking for an existing entry with this key,
    - we can recode it to be **O**(1) if we know key is new, since we can insert the new item at the beginning or at the end of the sequence
  - get and remove take **O**(***n***) time since in the worst case (item not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)
- We may want something faster…

Outline

- Map ADT (section 10.1)
- Sorted Map ADT (section 10.3)
- Binary Search Trees (section 11.1)
  - Definition
  - Searching
  - Operations on BSTs
  - Performance

Ordered map ADT

- Similar to the map ADT, but it also supports extra operations that are aware of an order between keys
  - eg “find key next above k”
  - Order can be defined by a given comparator for the entries
  - This allows for much wider categories of applications

Ordered map ADT (extra methods)

firstEntry(k): Returns the entry with smallest key value; if the map is empty, then it returns **null**.

lastEntry(k): Returns the entry with largest key value; if the map is empty, then it returns **null**.

ceilingEntry(k): Returns the entry with the least key value greater than or equal to k; if there is no such entry, then it returns **null**.

floorEntry(k): Returns the entry with the greatest key value less than or equal to k; if there is no such entry, then it returns **null**.

lowerEntry(k): Returns the entry with the greatest key value strictly less than k; if there is no such entry, then it returns **null**.

higherEntry(k): Returns the entry with the least key value strictly greater than k; if there is no such entry, then it returns **null**.

Ordered map implementation

- We can implement an ordered map using a sorted array
  - An “ordered search table”

Illustration shows keys only, but actually store key,value pairs

0	1	2	3	4	5	6	7	8	9	10
4	6	9	12	15	16	18	28	34		

Ordered search tables

- Space required is O(n)
- Insert and delete operations are O(n)
  - shift array entries up or down (recall from week 2)
- Search operations can be implemented efficiently using extended form of binary search (next slide).
  - This will give us a O(log n) upper bound on these operations.
- Ordered maps can also be implemented using Binary Search Trees (next topic)!

Binary search in sorted array without duplicates (recursive code, from textbook code fragment 10.11)

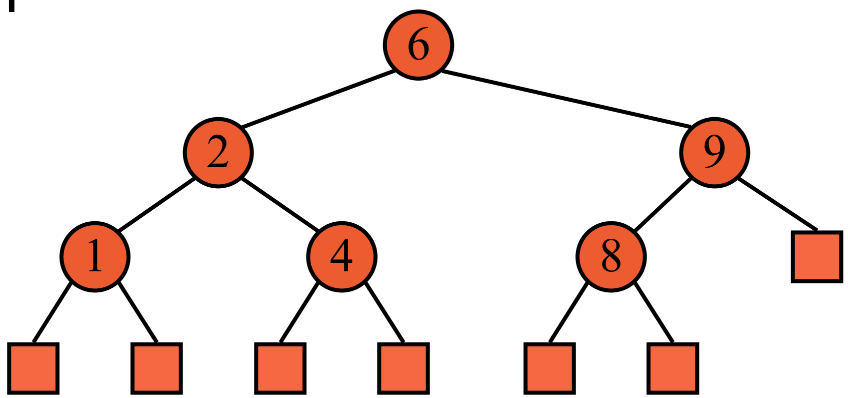
```
/** Return the smallest index
     for range table[low .. high] inclusive
     that is storing an entry with
     key greater than or equal to k
     (or else index high+1)
*/
int findIndex(K key, int low, int high) {
    if (high < low) return high+1;
    int mid = (high+low)/2;
    int comp = compare(key, table.get(mid));
    if (comp == 0) return mid;
    else if (comp < 0) return findIndex(key, low, mid-1);
    else return findIndex(key, mid+1, high);
}
```

Outline

- Map ADT (section 10.1)
- Sorted Map ADT (section 10.3)
- Binary Search Trees (section 11.1)
  - Definition
  - Searching
  - Operations on BSTs
  - Performance

Binary Search Trees

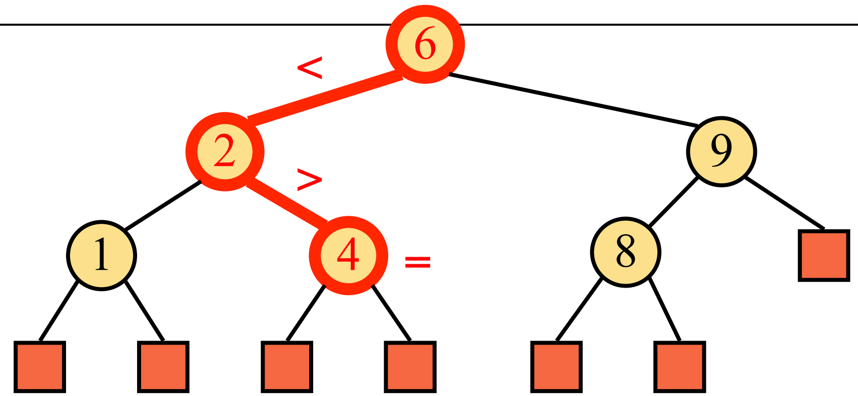
- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be any three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) < key(v) < key(w)$
- External nodes do not store items (and with careful coding, can be omitted, using null to refer to such)
- Therefore: An inorder traversal of a binary search trees visits the keys in increasing order



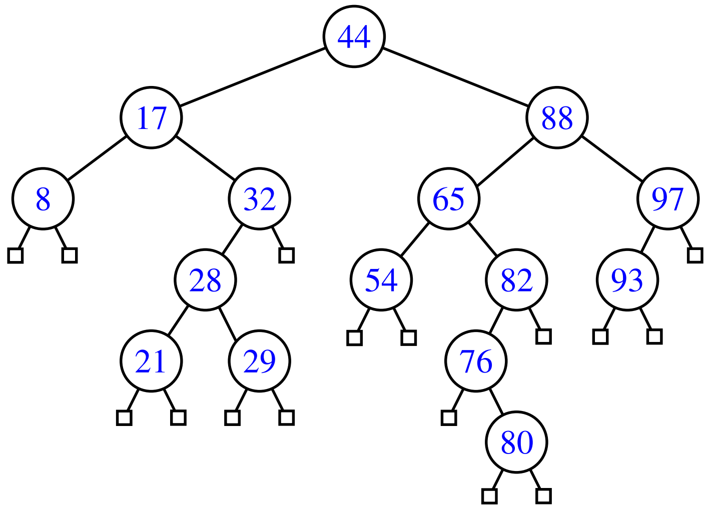
Searching with a Binary Search Tree

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach an external node, this means that the key is not found
- Example: searching for key 4:
  - Call TreeSearch(root,4)
- The algorithms for nearest neighbor queries are similar

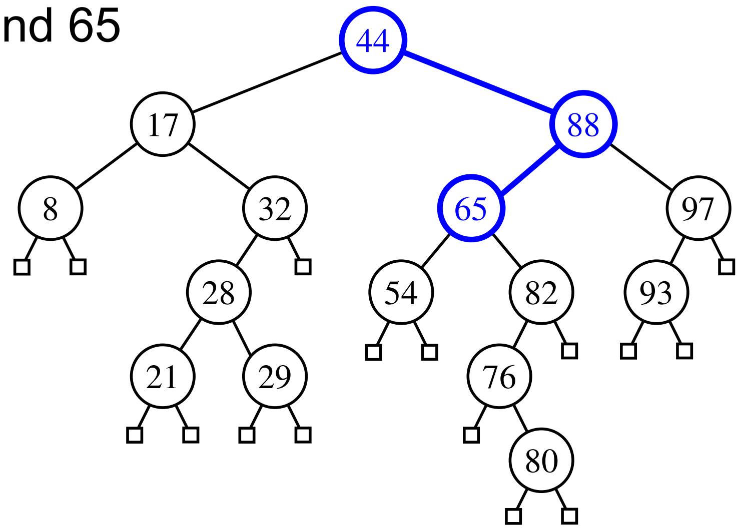
```
Algorithm TreeSearch(n, k)
    if n is external then
        return n        {unsuccessful search}
    if k == key(n) then
        return n        {successful search}
    else if k < key(v) then
        return TreeSearch (left(n),k)   {recur on
                                         left subtree}
    else                  {that is  k > key(v) }
        return TreeSearch(right(n),k) {recur on
                                         left subtree}
```



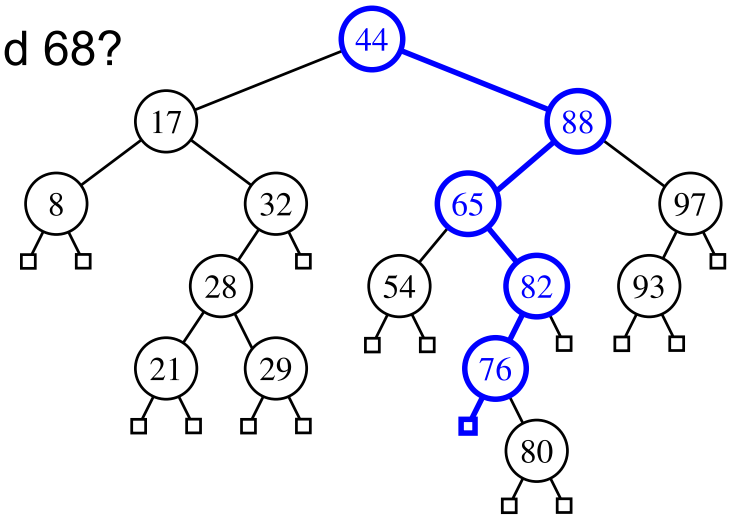
Example



Find 65

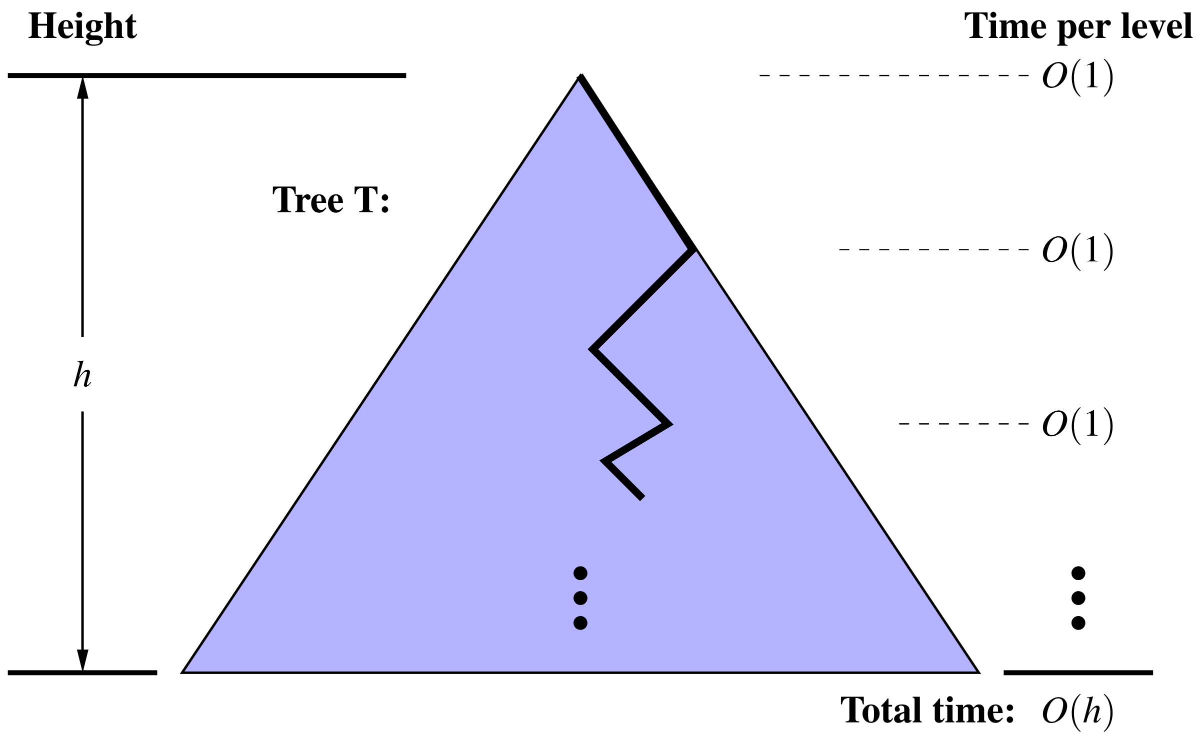


Find 68?



Analysis of Binary Tree Searching

- Runs in O(h) time, where h is the height of the tree
- Worst case is h = O(n) but there are “balanced tree” strategies to maintain h<= log(n)

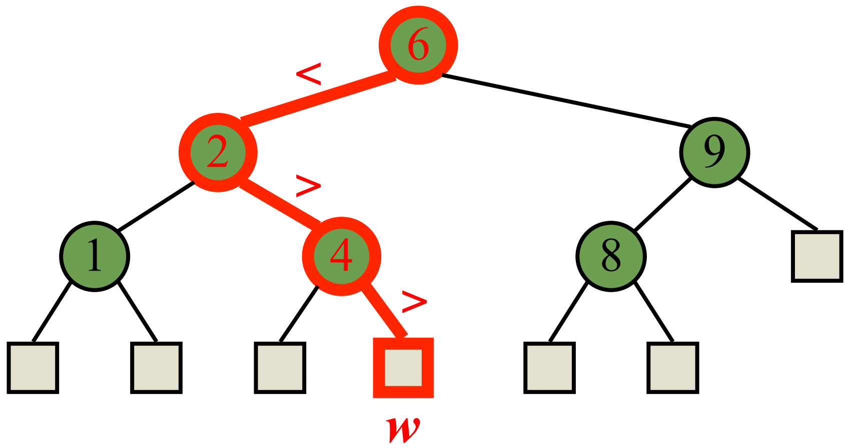


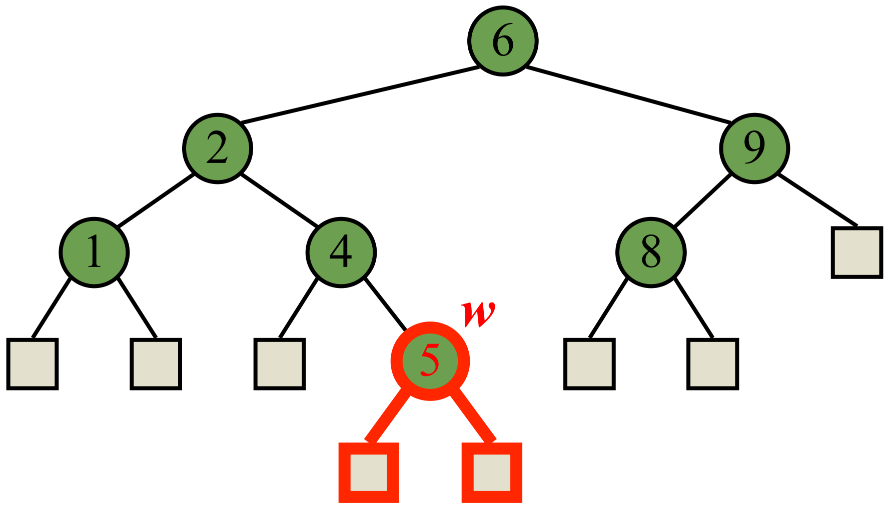
Insertion

To insert a new element with key k into the tree:

- If entry k already exists, replace it with the new value (note: there are other options)
- Otherwise, let w be the node that was reached at the end of the failed search, insert k at node w and expand w into an internal node
  - expandExternal(p,e) stores entry e at external position p, make p internal and add 2 new leaves as children
- Example: insert 5

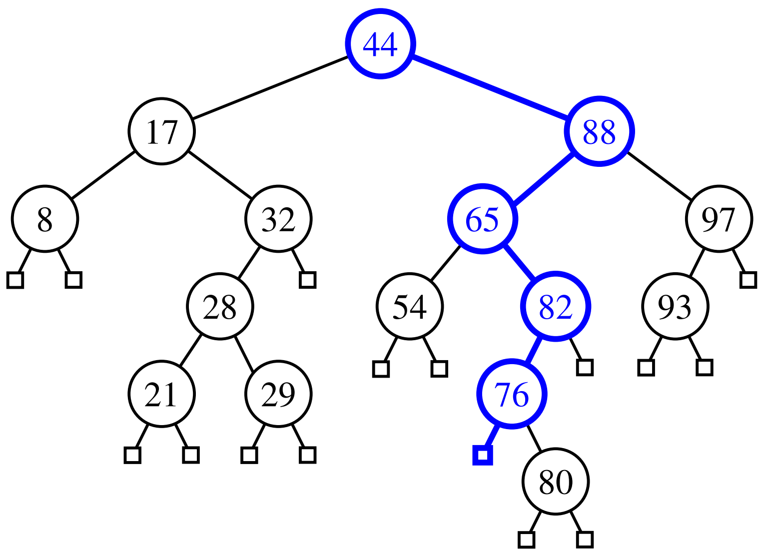
Algorithm ***TreeInsert*(*k*, *v*)**
{input: a search key to be associated with value v}
  p = TreeSearch (root(),k)
  if k == key(p) **then**
    change p's value to v
  else
    expandExternal(p,(k,v))

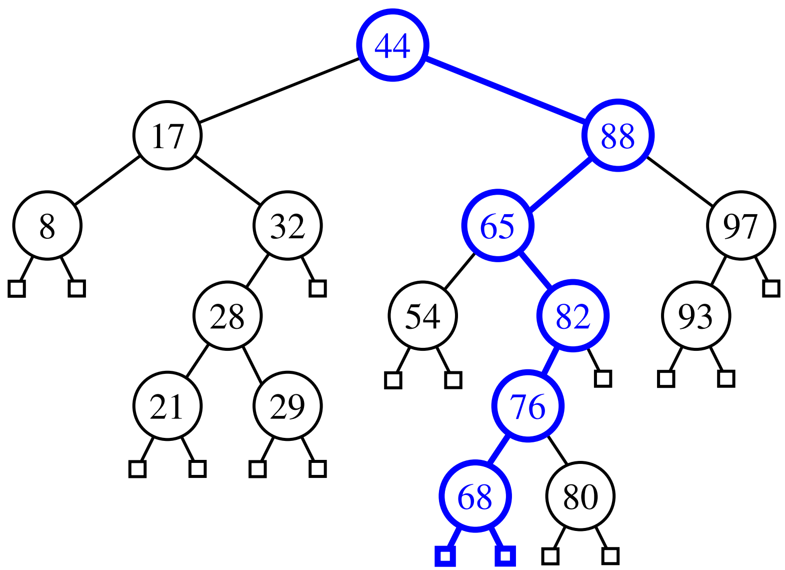




Example

Insert entry with key
68





In-class Exercise

Create the BST , inserting the following entries (key,value) in order:
(4, Joe),(7,Lucas),(8,Jane),(5,Emily),
(2,Bob),(6,Susan),(3,Henry)

What if they were instead inserted In order (2,Bob), (3,Henry),(4, Joe),
(5,Emily),(6,Susan),(7,Lucas),(8,Jane) ?

What if they were instead inserted In order (5,Emily) , (2,Bob), (3,Henry),(4, Joe), (6,Susan),(7, Lucas),(8, Jane) ?

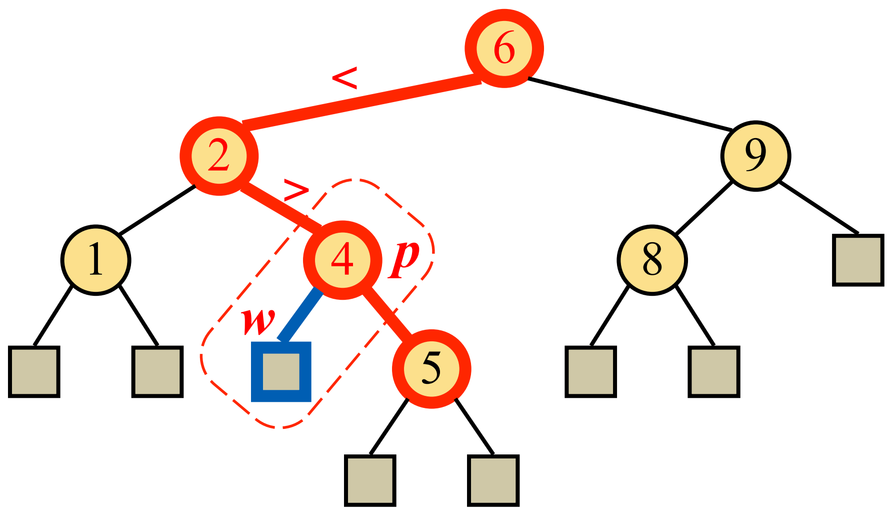
Deletion (1)

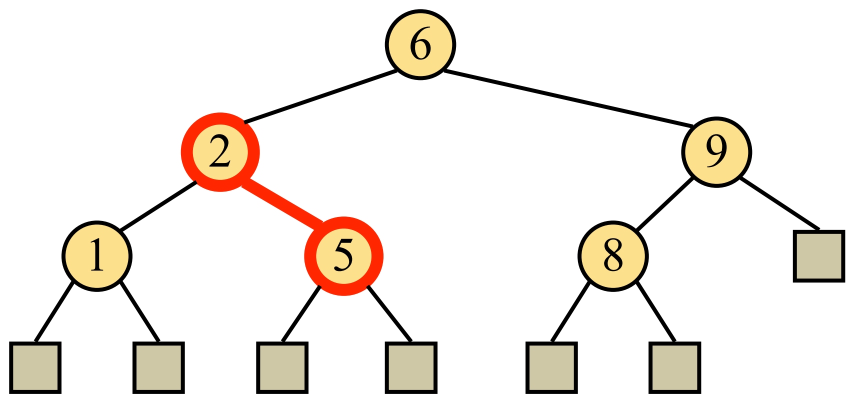
To delete an entry with key k from a BST:

(Assuming key $\boldsymbol{k}$ is in the tree, let $\boldsymbol{p}$ be the node storing $\boldsymbol{k}$)

CASE 1: If node $\boldsymbol{p}$ <u>has a leaf child $\boldsymbol{w}$</u>, we remove $\boldsymbol{p}$ and $\boldsymbol{w}$ from the tree with operation removeExternal($\boldsymbol{w}$), which removes $\boldsymbol{w}$ and its parent, and promote the other child upward to take p's place

- Example: remove 4

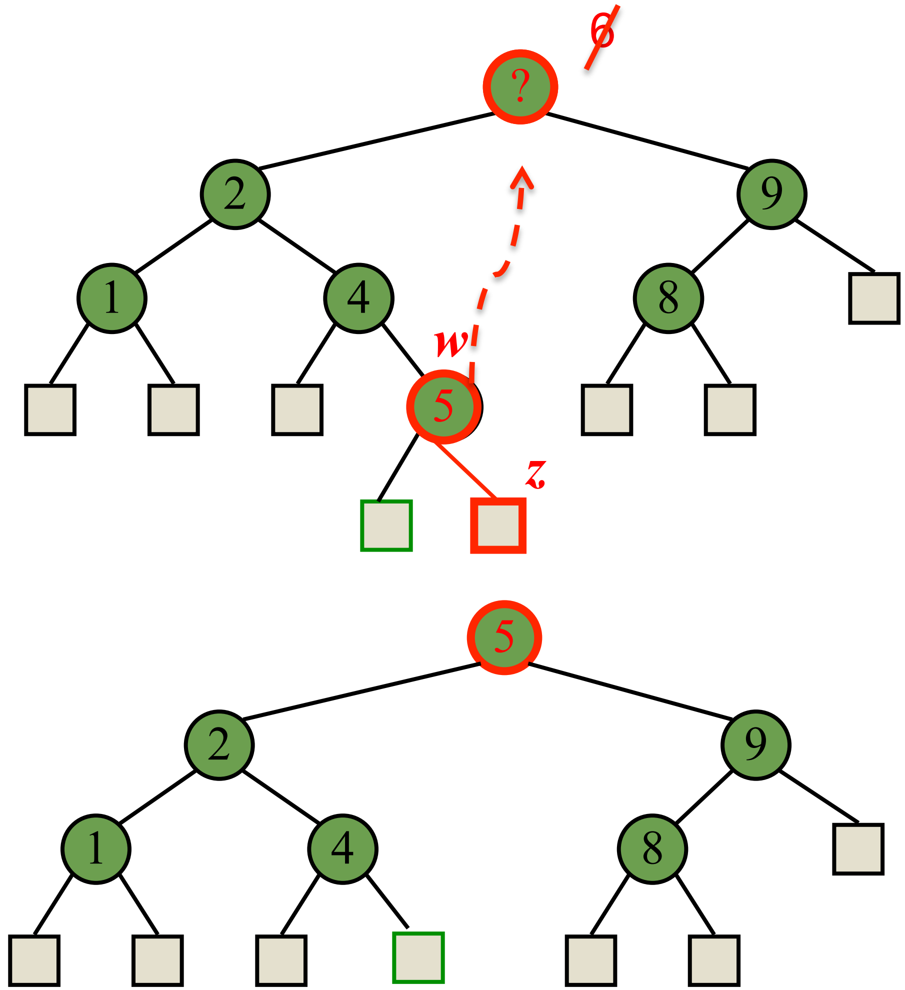




Deletion (2)

CASE 2: If node p's children are both internal (ex: 6),

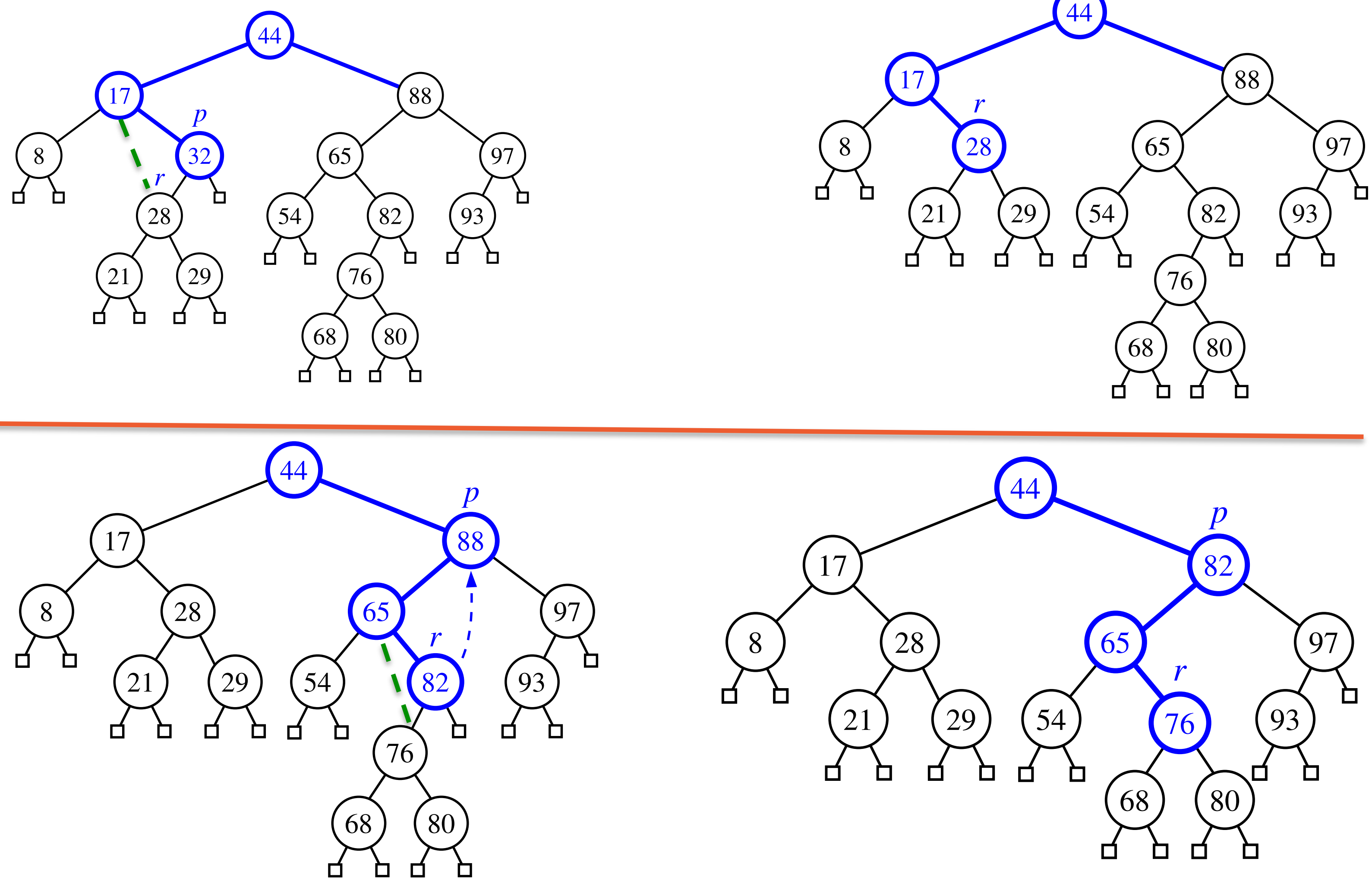
- we find the internal node w that immediately precedes p in an inorder traversal (5)
- we copy $key(w)$ into node p
- we remove node w and its right child z (which is always a leaf) with removeExternal(z)



Deletion algorithm

```
Algorithm TreeRemove(k)
    p = TreeSearch (root(),k)
    if p is external then return null {no key found}
    else if p has at least one child external w
        removeExternal(w)
    else {both p's children are internal}
        r = immediate predecessor of p  {right most internal position of p's left subtree}
        replace p with r
        TreeRemove(r)  {r only has no internal right children, so
```

Example

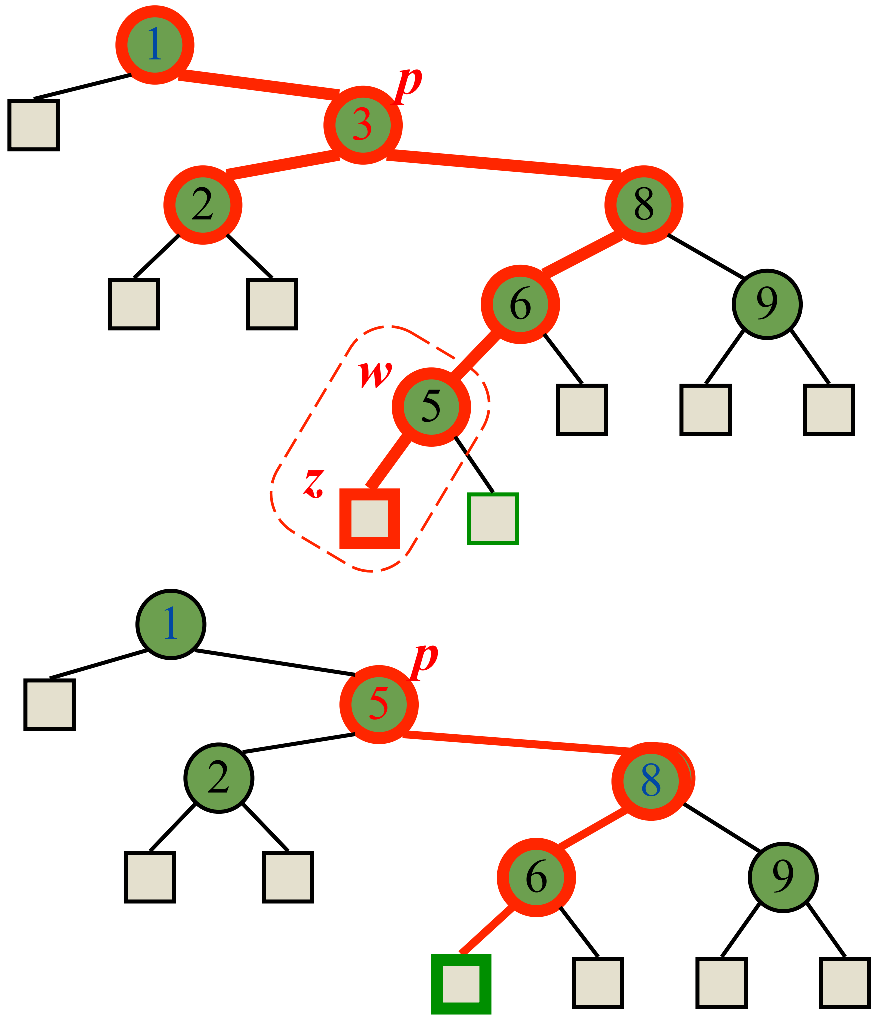


44 17 88 8 32 65 97 28 54 82 93 21 29 76 68 80
p
r
44 17 88 8 28 65 97 21 29 54 82 93 76 68 80
r
44 17 88 8 28 65 97 21 29 54 82 93 76 68 80
p
r
44 17 82 8 28 65 97 21 29 54 76 93 68 80
p
r

Deletion : another variant

- It is also possible to use the <u>smallest</u> key of the <u>right</u> subtree
  - we find the internal node w that immediately follows p in an inorder traversal
  - we copy $key(w)$ into node p
  - we remove node w and its left child z (which must be a leaf) with removeExternal(z)
- Example: remove 3

- Inorder successor/predecessor



Inorder pred/successors in BSTs

- Delete operation on an internal node with two children:
  - replace with <u>inorder predecessor</u>
  - Or, replace with <u>inorder successor</u>

- Finding inorder successors:
  - The inorder successor of a node v is the "left-most" node of the right subtree
  - The inorder successor is either a leaf or an internal node that has only a right child
    - Therefore inorder successors are "easy" to delete from a binary search tree
- Similarly for finding inorder predecessors
  - Right-most node of left subtree

Inorder successor

```
InorderNext(v):
    (if v has no right subtree, then there is no successor)
    temp = right child of v
    while (temp has a left child)
        temp = left child of temp
    return temp
```

Duplicate key values in BST

- Our definition says that keys are in strict increasing order

  ***key*(*left child*) < *key*(*parent*) < *key*(*right child*)**

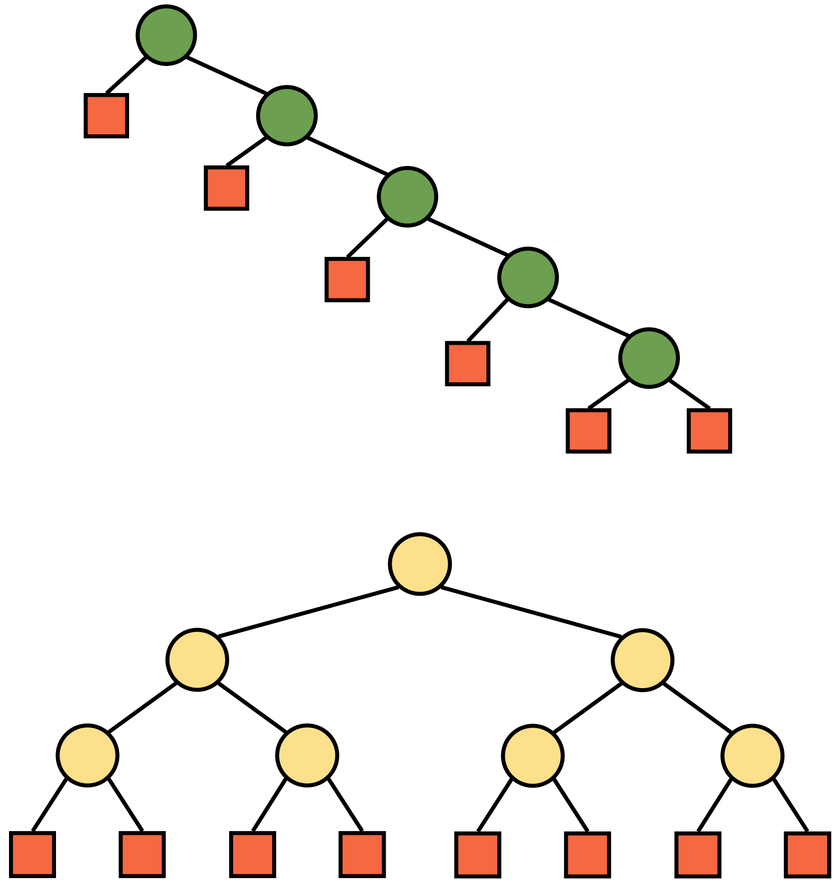
- This means that with this definition duplicate key values are <u>not</u> allowed.

- However, it is easy to change it to allow duplicates. But it means additional computational steps in the BST operations. Most common methods:
  - Allowing left children (or right children) to be equal to the parent node
    - ***key*(*left child*) ≤ *key*(*parent*) < *key*(*right child*)**
  - Using a list to store duplicates

recall: performance of a binary search tree

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

- Therefore insertions, removals, searching operations *hopes for* O(log n) but only guarantees O(n)



Keeping a search tree balanced

- We can design variants of binary search trees that:
  - always remain balanced enough, in order to
    - guarantee *a worst-case* search time of O(log *n)*
      - therefore what we need is to maintain at most O(log *n)* height
  - This is done with a binary search tree, but we do extra work on insert or delete, to reshape the tree and reduce its height
- Recall that a binary tree is balanced if the height of any node's right subtree differs from the height of the node's left subtree by no more than 1
- Algorithms are in textbook sections 11.2-11.6
  - some of these will be covered for INFO1905, as extra material

Summary

- Map ADT (section 10.1)
- Sorted Map ADT (section 10.3)
- Binary Search Trees (section 11.1)
  - Definition
  - Searching
  - Operations on BSTs
  - Performance

These algorithms are a vital skill for the exam