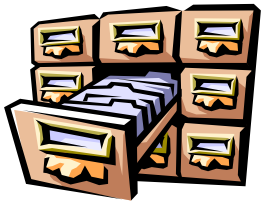
Maps and Dictionaries



Map and Dictionary ADTs

The dictionary ADT models a searchable collection of key-element items

The main operations of a dictionary are searching, inserting, and deleting items

MAP: Multiple items with the same key are NOT allowed

DICTIONARY: Multiple items with the same key ARE allowed

Applications:

- address book
- student-record database
- credit card authorization

Dictionary ADT methods:

find(k): if the dictionary has an entry with key k, returns the entry; otherwise, returns null

insert(k, o): inserts an entry (k, o) into the dictionary

remove(e): removes entry e from the dictionary and returns the entry; otherwise, returns null if e not found

size(), isEmpty()

entries(): returns the key-value entries stored

findAll(k): returns an iterator of all entries with key k

Map ADT methods:

get(k): if the map has an entry with key k, returns the entry; otherwise, returns null

put(k, o): adds an entry (k, o) into the map if it doesn't exist; otherwise, the old value is replaced

remove(k): removes entry with key k from the map and returns its value; otherwise, returns null if k not found

size(), isEmpty()

keys(): returns an iterator of keys stored in the map

values(): returns an iterator of values associated with keys stored in the map

Dictionary ADTs

- Ordered dictionary: total order relation defined by some comparator for the keys
- Unordered dictionary: no order relation defined for keys

Implementing a Dictionary with an Unordered Sequence

- ***unordered sequence***

- searching and removing takes O(n) time
- inserting takes O(1) time
- applications to log files (frequent insertions, rare searches and removals)

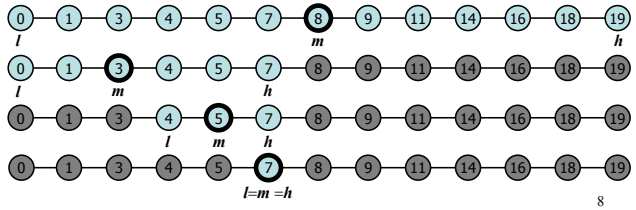
Implementing a Dictionary with an Ordered Sequence

- ***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)

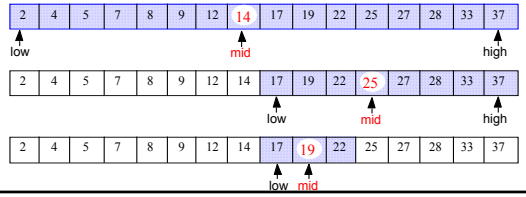
Binary Search

- narrow down the search range in stages
- "high-low" game
- Example: find(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

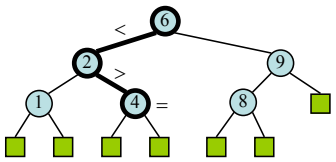
- The range of candidate items to be searched is ***divided into half after each comparison***

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

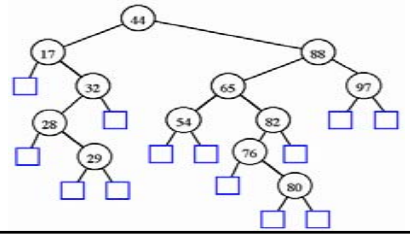
Binary Search Trees

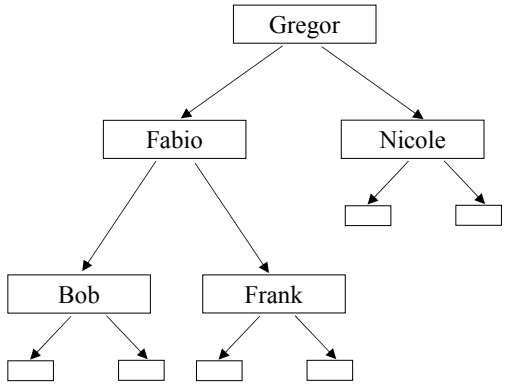
- Searching
- Cost of Searching
- Insertion
- Deletion

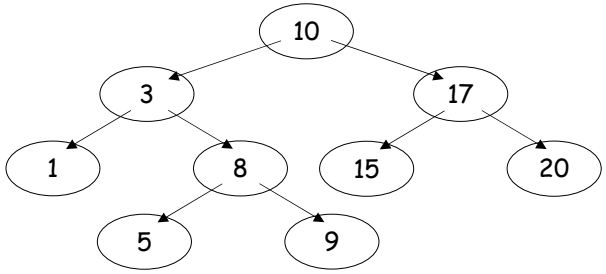


Binary Search Trees

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







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

Operations

Searching: find(k)

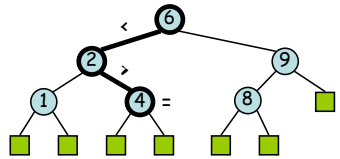
Inserting: insert(k, o)

Removing: remove(k)

Search

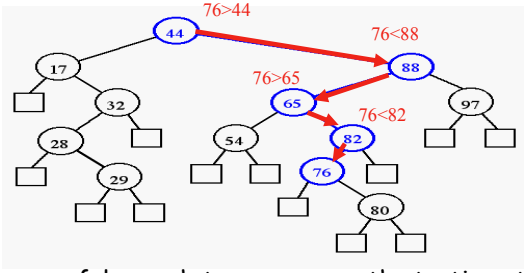
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY
- Example: find(4)
- call TreeSearch(4,root)

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



Search Example I

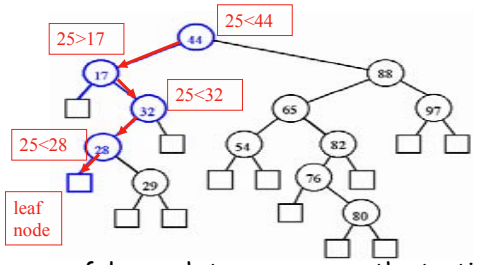
Successful find(76)



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

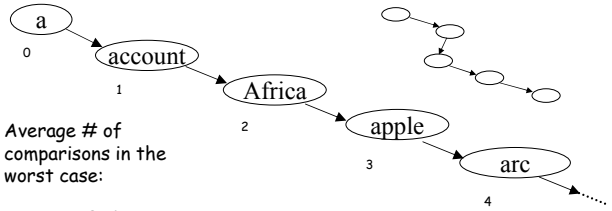
Search Example II

Unsuccessful find(25)



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

Cost of Search: Worst Case



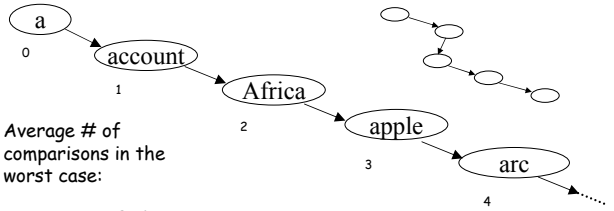
Average # of comparisons in the worst case:

Successful search

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

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

Cost of Search: Worst Case

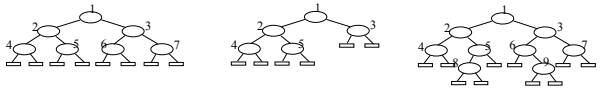


Average # of comparisons in the worst case:

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 = $\lfloor \log i \rfloor$

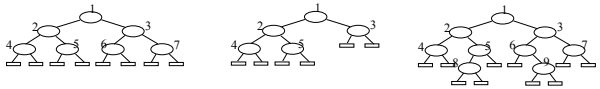
For a **successful** search, we do 2 comparisons at each node along the path plus one at the end.

Comparisons to node i: $O(\log i)$

→ Average # of comparisons in the best case

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

Cost of Search: Best Case



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

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

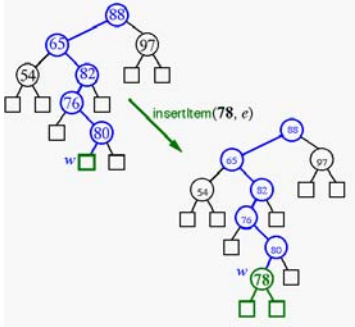
For a **failed** search, we do 2 comparisons at each node along the path plus two at the end.

Only paths to external nodes count.

So, always $O(\log n)$

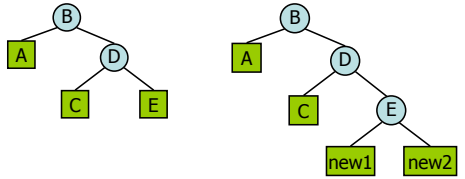
Insertion I

- To perform insert(k,e), call TreeInsert(k, e, T.root())
- 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 insertAtExternal(w) on T and store (k, e) in w



insertAtExternal(v):

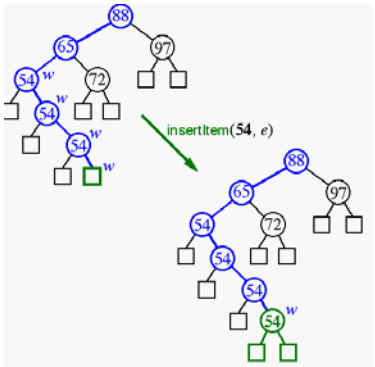
Transform v from an external node into an internal node by creating two new children



insertAtExternal(v):
new1 and new 2 are the new nodes
if isExternal(v)
  v.left ← new1
  v.right ← new2
  size ← size +2

Insertion II

- If w is internal, we know another item with key k is stored at w. We call the algorithm recursively starting at T.right(w) or T.left(w)
- The idea is to store the new item in an external node which either precedes or follows the items with the same key in an inorder traversal.



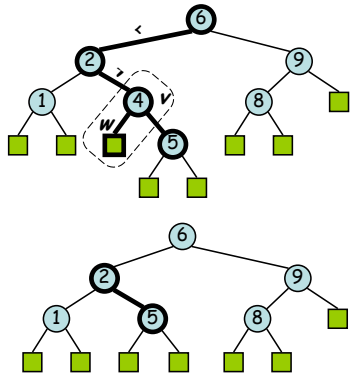
Construct a Tree

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

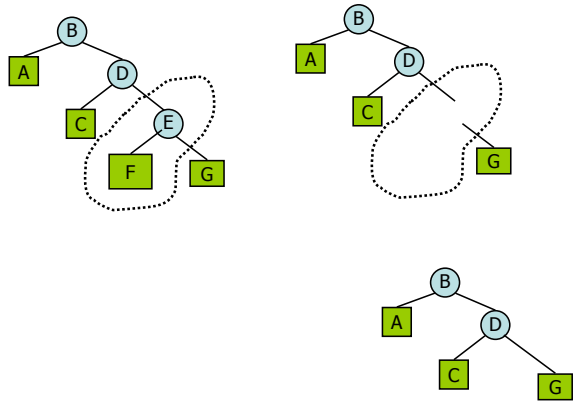
a. 5,8,3,7,1,9,2,4,6
b. 1,2,3,4,5,6,7,8,9
c. 5,4,6,3,7,2,8,1,9

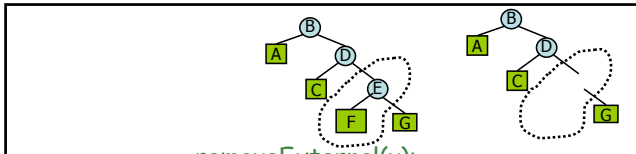
Deletion I

- 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 removeExternal(*w*)
- Example: remove 4

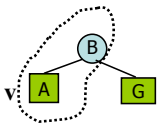


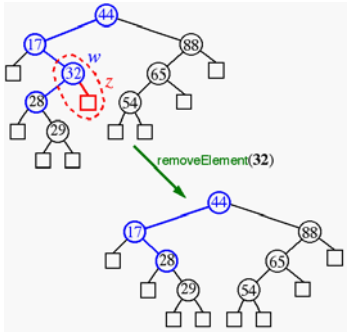
removeExternal(v):





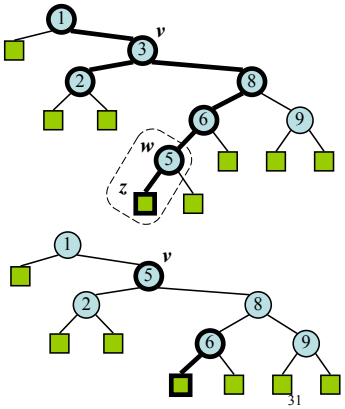

```
removeExternal(v):
if isExternal(v)
   { p ← parent(v)
     s ← sibling(v)
     if isRoot(v)  s.parent  ←  null and root ← s
     else
       { g ← parent(p)
         if (p is leftChild(g)  g.left ← s
              else g.right ← s
         s.parent ← g
         }
     size ← size - 2 }
```





Deletion II

- We consider the case where the key *k* to be removed is stored at a node *v* whose children are both internal
  - we find the internal node *w* that follows *v* in an inorder traversal
  - we copy *key*(*w*) into node *v*
  - we remove node *w* and its left child *z* (which must be a leaf) by means of operation removeExternal(*z*)
- Example: remove 3



Practice, practice, practice…

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

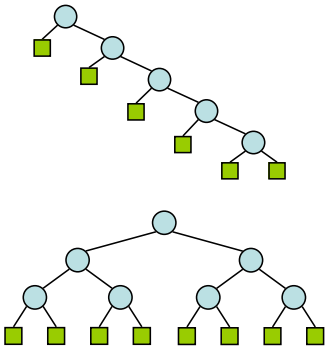
Cost of Inserting and Deleting = Cost of Search

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 *find*, *insert* and *remove* 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 will be explored next.