

CS61B Lecture #20

Administrative:

Today:

- Search Trees
- Priority Queues and Heaps

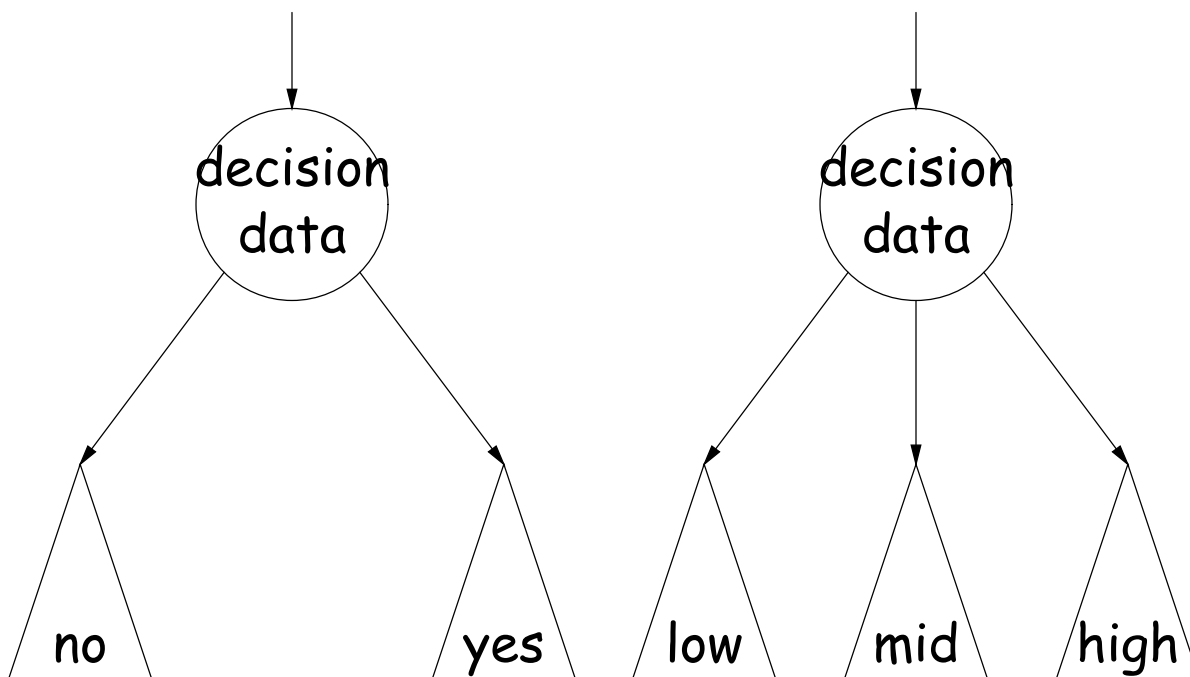
Readings for Today: *Data Structures, Chapter 6*

Readings for Next Topic:

Data Structures, Chapter 7 (Hashing)

Divide and Conquer

- Much (most?) computation is devoted to finding things in response to various forms of query.
- Linear search for response can be expensive, especially when data set is too large for primary memory.
- Preferable to have criteria for *dividing* data to be searched into pieces recursively
- Remember figure for $\lg N$ algorithms: at $1\mu\text{sec}$ per comparison, could process 10^{300000} items in 1 sec.
- Tree is a natural framework for the representation:



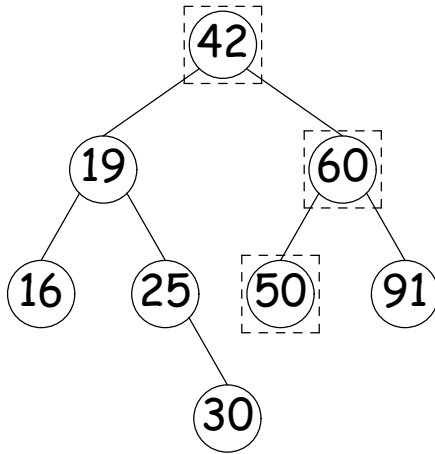
Binary Search Trees

Binary Search Property:

- Tree nodes contain *keys*, and possibly other data.
- All nodes in left subtree of node have *smaller keys*.
- All nodes in right subtree of node have *larger keys*.
- "Smaller" means any complete transitive, anti-symmetric ordering on keys:
 - exactly one of $x \prec y$ and $y \prec x$ true.
 - $x \prec y$ and $y \prec z$ imply $x \prec z$.
 - (To simplify, won't allow duplicate keys this semester).
- E.g., in dictionary database, node label would be (*word, definition*): *word* is the key.

Finding

- Searching for 50 and 49:

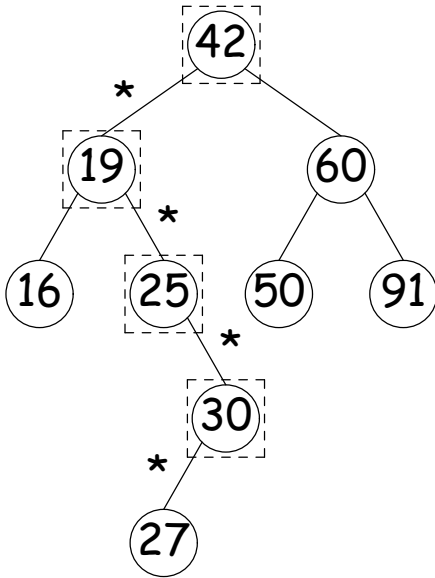


```
/** Node in T containing L,  
 * or null if none */  
static BST find(BST T, Object L) {  
    if (T == null)  
        return T;  
    if (L.keyequals (T.label()))  
        return T;  
    else if (L < T.label())  
        return find(T.left(), L);  
    else  
        return find(T.right (), L);  
}
```

- Dashed boxes show which node labels we look at.
- Number looked at proportional to height of tree.

Inserting

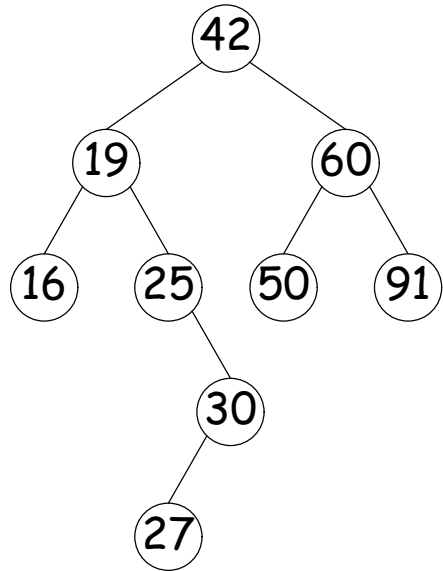
- Inserting 27



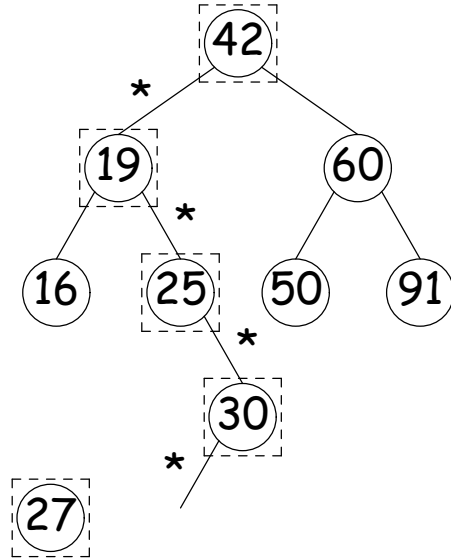
```
/** Insert L in T, replacing existing
 * value if present, and returning
 * new tree. */
BST insert(BST T, Object L) {
  if (T == null)
    return new BST(L);
  if (L.keyequals (T.label()))
    T.setLabel (L);
  else if (L < T.label())
    T.setLeft(insert (T.left (), L));
  else
    T.setRight(insert (T.right (), L));
  return T;
}
```

- Starred edges are set (to themselves, unless initially null).
- Again, time proportional to height.

Deletion

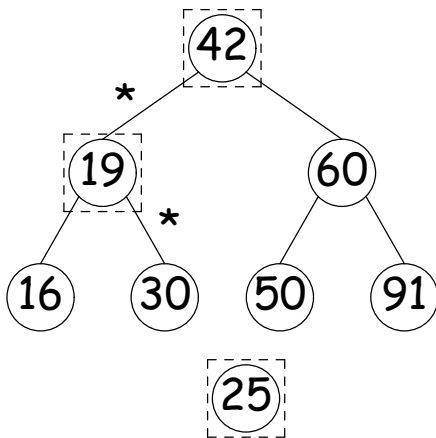


Initial

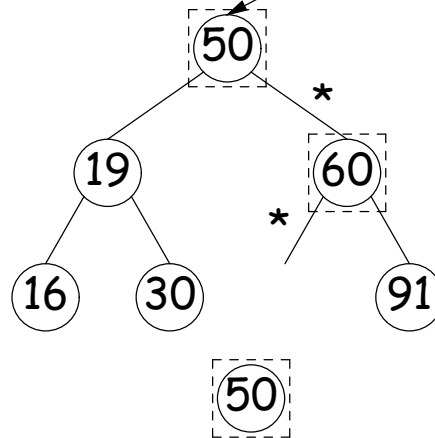


Remove 27

formerly contained 42



Remove 25



Remove 42

Problems

- For “bushy” tree, height near $\lg N$, and so are search, insertion, deletion times
- But as tree gets unbalanced (list-like), these times go to N .
- We’ll deal with balance of search trees later.
- However, if binary search property can be relaxed, balance is easier.

Priority Queues, Heaps

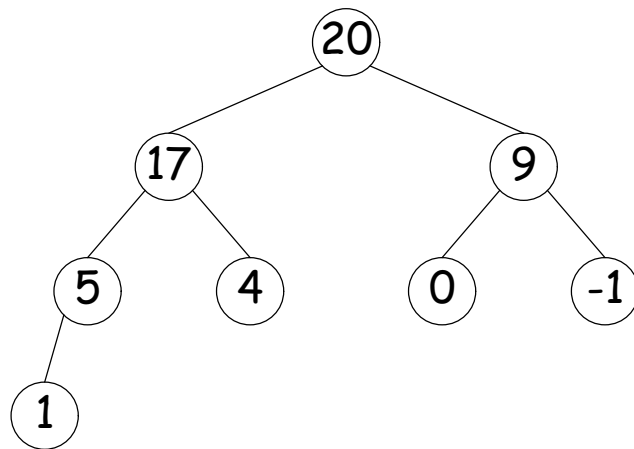
- Priority queue: defined by operations "add," "find largest," "remove largest."
- Examples: scheduling long streams of actions to occur at various future times.
- Also useful for sorting (keep removing largest).
- Heap is common implementation.
- Enforces *heap property*: all labels in *both* children of node are less (or greater) than node's label.
- So node at top has largest (or smallest) label.
- Can add smaller value in less bushy subtree, thus keeping bushiness balanced.
- Insertion and deletion always proportional to $\lg N$ in worst case.

Example: Inserting into a simple heap

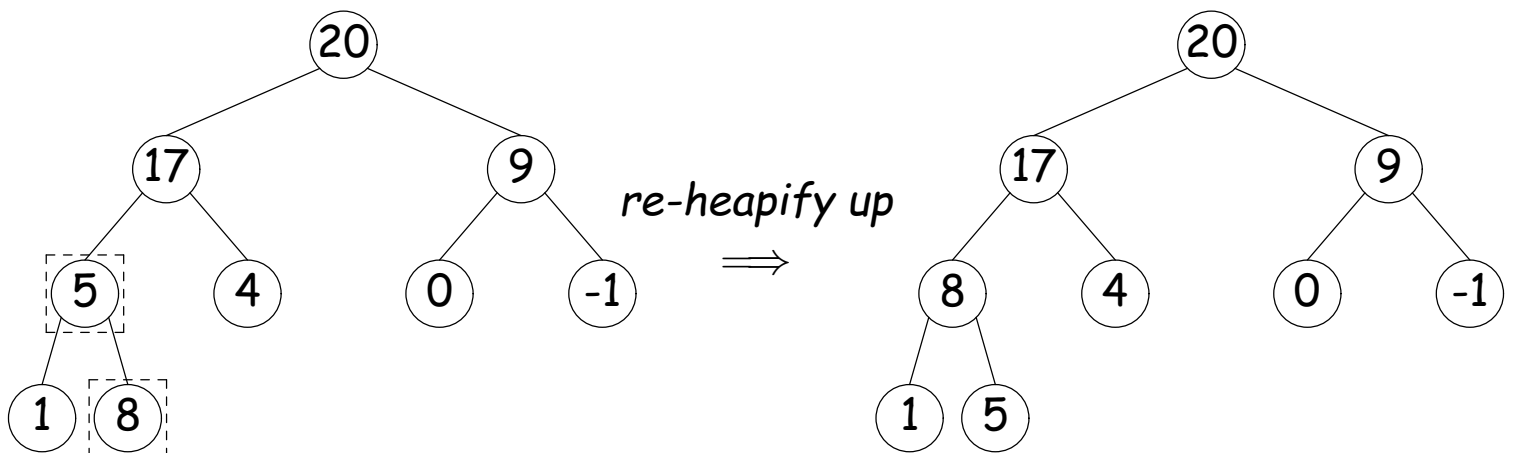
Data:

1 17 4 5 9 0 -1 20

Initial Heap:

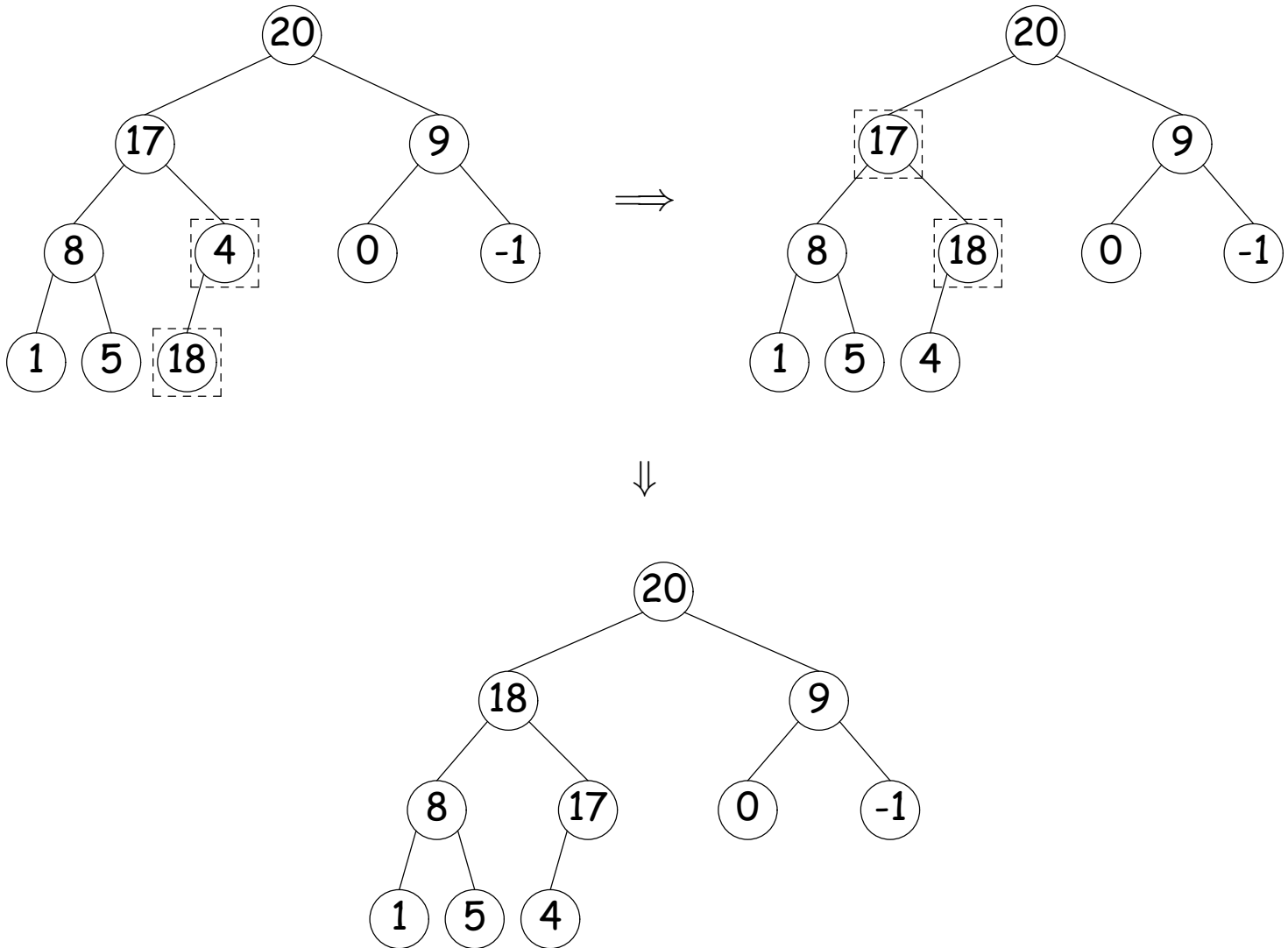


Add 8: Dashed boxes show where heap property violated



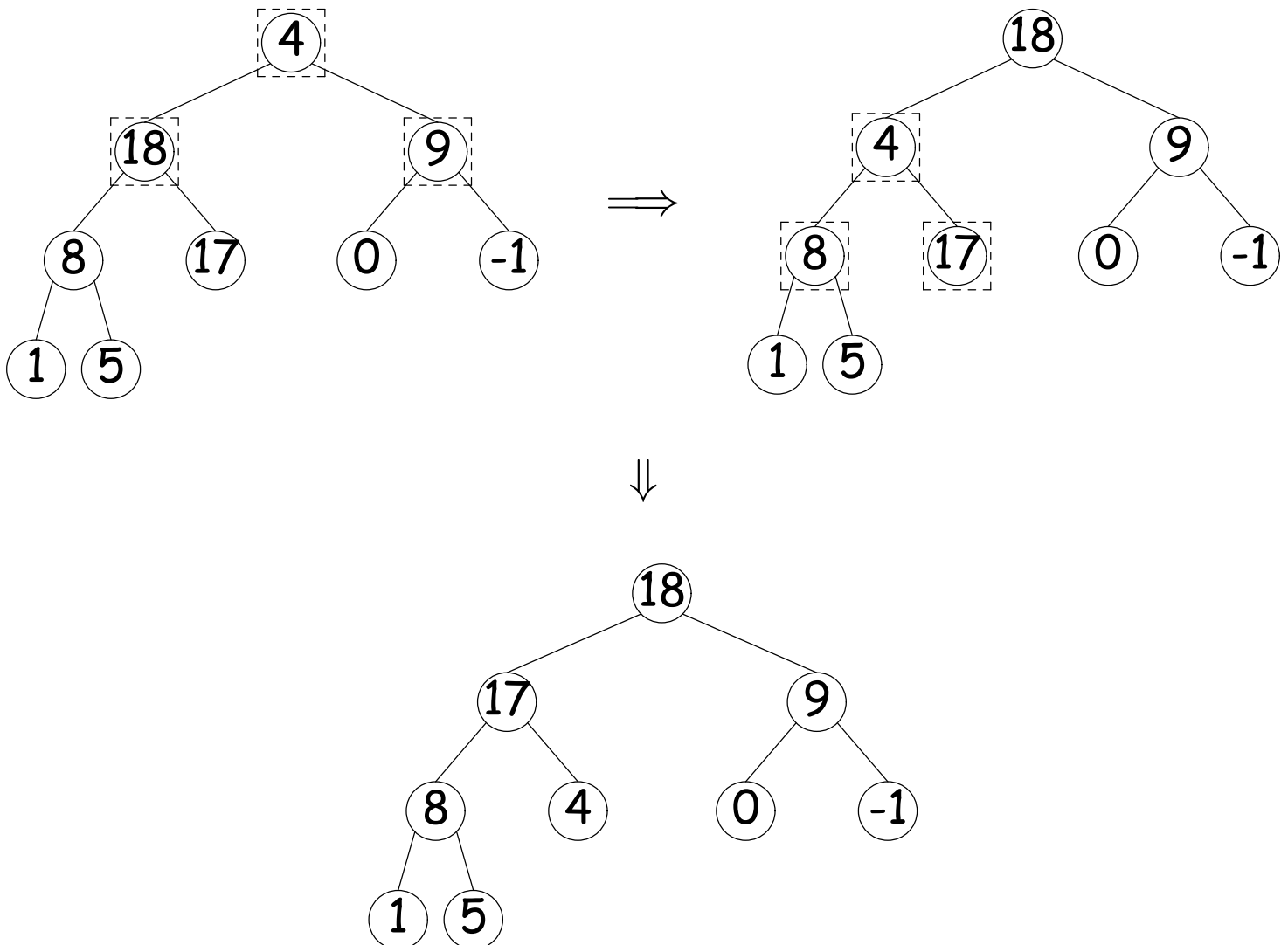
Heap insertion continued

Now insert 18:



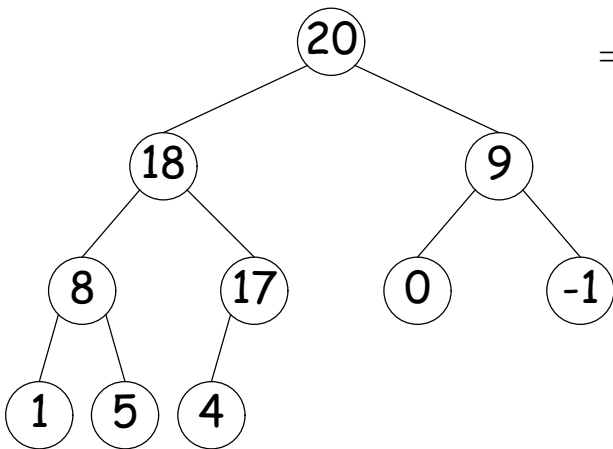
Removing Largest from Heap

Now remove largest: Move bottommost, rightmost node to top, then re-heapify down as needed (swap offending node with larger child) to re-establish heap property.



Heaps in Arrays

- Since heaps are complete, missing items only at bottom level, can use arrays for compact representation.
- Example of removal from last slide (dashed arrows show children):



⇒

