

## CS61B Lecture #23

### Administrative:

- Midterm grades posted (mean: 26, median: 27)
- Will be available from TAs next week.

### Today:

- Search Trees
- Priority Queues and Heaps

**Readings for Today:** *Data Structures*, Chapter 6, Goodrich & Tamassia, Chapters 6, 7, and §9.1.

### Readings for Next Topic:

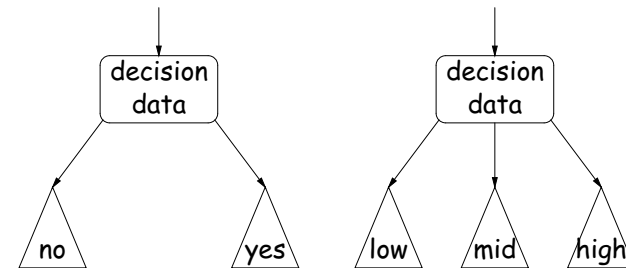
*Data Structures*, Chapter 7 (Hashing), Goodrich & Tamassia, §8.1-8.3.

Last modified: Mon Oct 21 02:23:23 2002

CS61B: Lecture #23 1

## 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:



Last modified: Mon Oct 21 02:23:23 2002

CS61B: Lecture #23 2

## Binary Search Trees

### Binary Search Property:

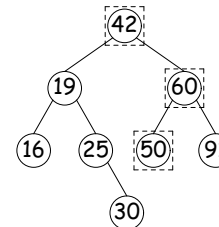
- Tree nodes contain *keys*, and possibly other data.
- All nodes in left subtree of node have *smaller keys*.
- All nodes in right subtree of node have *larger keys*.
- "Smaller" means any complete transitive, anti-symmetric ordering on keys:
  - exactly one of  $x < y$  and  $y < x$  true.
  - $x 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

Last modified: Mon Oct 21 02:23:23 2002

CS61B: Lecture #23 3

## 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);
}
```

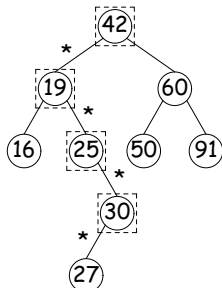
- Dashed boxes show which node labels we look at.
- Number looked at proportional to height of tree.

Last modified: Mon Oct 21 02:23:23 2002

CS61B: Lecture #23 4

## Inserting

### • Inserting 27



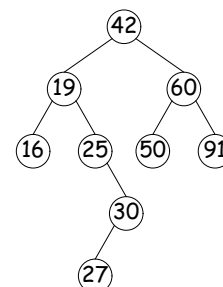
```

/** 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;
}

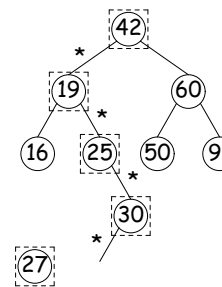
```

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

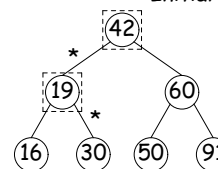
## Deletion



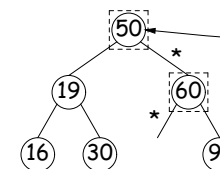
Initial



Remove 27



Remove 25



Remove 42

← formerly contained 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

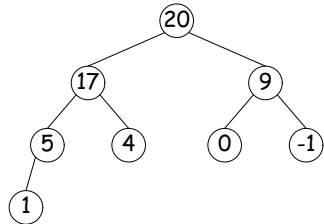
- Priority queue: defined by operations “add,” “find largest,” “remove largest.”
- Examples: scheduling long streams of actions to occur at various future times.
- Also useful for sorting (keep removing largest).
- Heap is common implementation.
- Enforces *heap property*: all labels in *both* children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

## Example: Inserting into a simple heap

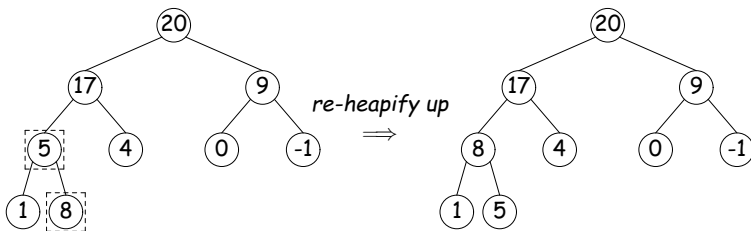
Data:

1 17 4 5 9 0 -1 20

Initial Heap:



Add 8: Dashed boxes show where heap property violated

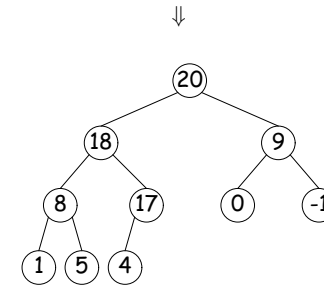
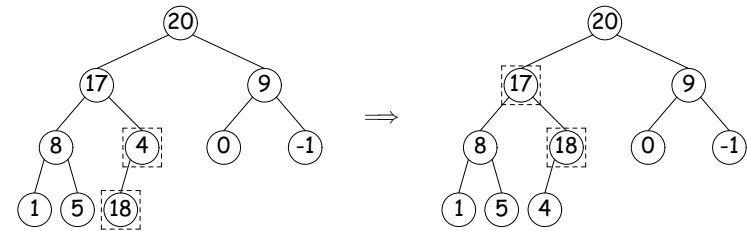


Last modified: Mon Oct 21 02:23:23 2002

CS61B: Lecture #23 9

## Heap insertion continued

Now insert 18:

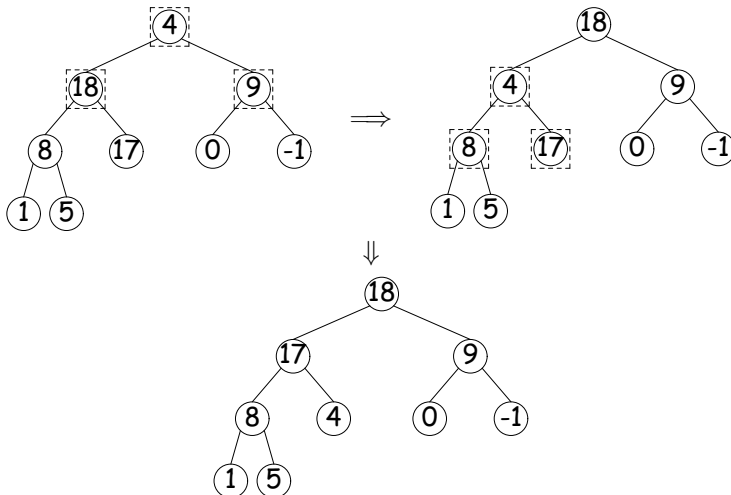


Last modified: Mon Oct 21 02:23:23 2002

CS61B: Lecture #23 10

## Removing Largest from Heap

To 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.

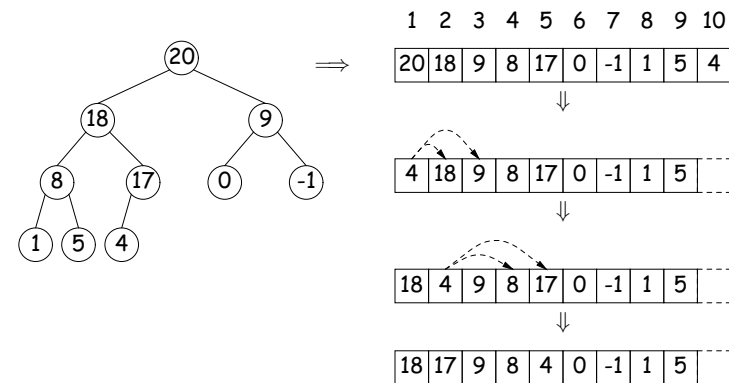


Last modified: Mon Oct 21 02:23:23 2002

CS61B: Lecture #23 11

## 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):



Last modified: Mon Oct 21 02:23:23 2002

CS61B: Lecture #23 12