## Priority Queues, Heaps

le: defined by operations "add," "find largest," "remove

cheduling long streams of actions to occur at various

or sorting (keep removing largest).

ementation is the heap, a kind of tree.

this same term is used to described the pool of storage operator uses. Sorry about that.)

17:56 2018 CS61B: Lecture #23 2

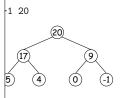
#### CS61B Lecture #23

es (Data Structures §6.4, §6.5) s (§6.2)

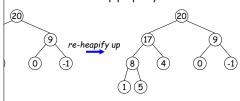
S: SortedSet, Map, etc.

Hashing (Data Structures Chapter 7).

### imple: Inserting into a simple heap



boxes show where heap property violated



17:56 2018 CS61B: Lecture #23 4

### Heaps

is a binary tree that enforces the

perty: Labels of both children of each node are less is label.

pp has largest label.

binary search property, which allows us to keep tree

always valid to put the smallest nodes anywhere at the

can be made *nearly complete*: all but possibly the last many keys as possible.

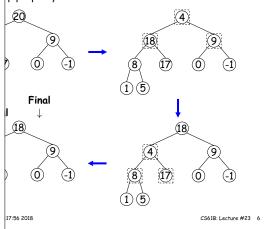
insertion of new value and deletion of largest value alme proportional to  $\lg N$  in worst case.

s basically the same, but with the minimum value at the dren having larger values than their parents.

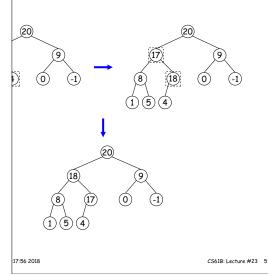
17:56 2018 CS61B: Lecture #23 3

# Removing Largest from Heap

gest: Move bottommost, rightmost node to top, then as needed (swap offending node with larger child) to ap property.



## Heap insertion continued



17:56 2018 CS61B: Lecture #23 1

### Ranges

looked for specific items

s, need an ordering anyway, and can also support looking fixalues.

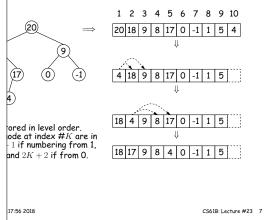
rform some action on all values in a BST that are within in natural order):

```
TTODO to all labels in T that are >= L and < U,
ing natural order. */
isitRange(BST<String> T, String L, String U,
          Consumer<BST<String>> whatToDo) {
11) {
eft = L.compareTo(T.label ()),
ight = U.compareTo(T.label ());
                           /* L < label */
eft < 0)
nge (T.left(), L, U, whatToDo);
eft <= 0 && compRight > 0) /* L <= label < U */
ight > 0)
                           /* label < U */
nge (T.right (), L, U, whatToDo);
17:56 2018
                                             CS61B: Lecture #23 8
```

## Heaps in Arrays

are nearly complete (missing items only at bottom level), vs for compact representation.

emoval from last slide (dashed arrows show children):



### red Sets and Range Queries in Java

Set supports range queries with views of set:

```
t(U): subset of S that is < U.
 t(L): subset that is \ge L.
```

(L,U): subset that is  $\geq$  L, < U.

iews modify S.

, e.g., add to a headSet beyond U are disallowed.

hrough a view to process a range:

```
String> fauna = new TreeSet<String>
ys.asList ("axolotl", "elk", "dog", "hartebeest", "duck"));
yitem: fauna.subSet ("bison", "gnu"))
out.printf ("%s, ", item);
dog, duck, elk,"
```

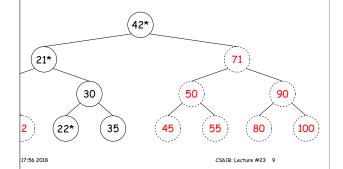
17:56 2018 CS61B: Lecture #23 10

# Time for Range Queries

ge query  $\in O(h+M)$ , where h is height of tree, and M data items that turn out to be in the range.

rching the tree below for all values  $25 \le x < 40$ .

s are never looked at. Starred nodes are looked at but he h comes from the starred nodes; the M comes from n-dashed nodes.



### ample of Representation: BSTSet

```
entation for SortedSet<String>
                    fauna = new BSTSet<String>(stuff);
d subsets.
                    subset1 = fauna.subSet("bison","gnu");
 BST, plus
                    subset2 = subset1.subSet("axolotl","dog");
kpensive!
                                              sentinel
                                   hartebeest
        hison
                             dog
         gnu
                                      elk
                    axolotl
        bison
         dog
                               duck
17:56 2018
                                              CS61B: Lecture #23 12
```

#### **TreeSet**

type TreeSet<T> requires either that T be Comparable, provide a Comparator, as in:

String> rev\_fauna = new TreeSet<String>(Collections.reverseOrder());

#### is a type of function object:

```
comparator<T> {
turn <0 if LEFT<RIGHT, >0 if LEFT>RIGHT, else 0. */
spare(T left, T right);
```

ith what Comparator<T extends Comparable<T>> is all

the reverseOrder comparator is defined like this:

```
arator that gives the reverse of natural order. */
extends Comparable<T>> Comparator<T> reverseOrder() {
  figures out this lambda expression is a Comparable<T>.
  (x, y) -> y.compareTo(x);
```

17:56 2018 CS61B: Lecture #23 11