Today:

- Priority queues (*Data Structures* §6.4, §6.5)
- Range queries (§6.2)
- Java utilities: `SortedSet`, `Map`, etc.

Next topic: Hashing (*Data Structures* Chapter 7).
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).

• Common implementation is the heap, a kind of tree.

• (Confusingly, this same term is used to described the pool of storage that the new operator uses. Sorry about that.)
Heaps

- A **max-heap** is a binary tree that enforces the
  
  **Heap Property:** Labels of both children of each node are less than node’s label.

- So node at top has largest label.

- Looser than binary search property, which allows us to keep tree “bushy”.

- That is, it’s always valid to put the smallest nodes anywhere at the bottom of the tree.

- Thus, heaps can be made **nearly complete:** all but possibly the last row have as many keys as possible.

- As a result, insertion of new value and deletion of largest value always take time proportional to $\lg N$ in worst case.

- A **min-heap** is basically the same, but with the minimum value at the root and children having larger values than their parents.
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

re-heapify up
Heap insertion continued

Now insert 18:

Before:

20
  17
  9
  8
  0
  1
  5
  18

After:

20
  17
  9
  8
  0
  1
  5
  18
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.

![Initial heap comparison](image-url)
Heaps in Arrays

- Since heaps are nearly complete (missing items only at bottom level), can use arrays for compact representation.

- Example of removal from last slide (dashed arrows show children):

  \[
  \begin{array}{cccccccccc}
  \textbf{20} & \textbf{18} & \textbf{9} & \textbf{8} & \textbf{17} & 0 & -1 & 1 & 5 & 4 \\
  \textbf{4} & 18 & 9 & 8 & 17 & 0 & -1 & 1 & 5 & 4 \\
  \textbf{18} & 17 & 9 & 8 & 4 & 0 & -1 & 1 & 5 & 4 \\
  \end{array}
  \]

Nodes stored in level order. Children of node at index \(\#K\) are in \(2K\) and \(2K+1\) if numbering from 1, or \(2K+1\) and \(2K+2\) if from 0.
Ranges

• So far, have looked for specific items

• But for BSTs, need an ordering anyway, and can also support looking for **ranges of values**.

• Example: perform some action on all values in a BST that are within some range (in natural order):

```java
/** Apply WHATTODO to all labels in T that are >= L and < U, in ascending natural order. */
static void visitRange(BST<String> T, String L, String U,
                        Consumer<BST<String>> whatToDo) {
    if (T != null) {
        int compLeft = L.compareTo(T.label ()),
                    compRight = U.compareTo(T.label ());
        if (compLeft < 0) /* L < label */
            visitRange (T.left(), L, U, whatToDo);
        if (compLeft <= 0 && compRight > 0) /* L <= label < U */
            whatToDo.accept(T);
        if (compRight > 0) /* label < U */
            visitRange (T.right (), L, U, whatToDo);
    }
}
```
Time for Range Queries

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

- Consider searching the tree below for all values $25 \leq x < 40$.

- Dashed nodes are never looked at. Starred nodes are looked at but not output. The $h$ comes from the starred nodes; the $M$ comes from unstared non-dashed nodes.
Ordered Sets and Range Queries in Java

- **Class** `SortedSet` supports range queries with *views* of set:
  - `S.headSet(U)`: subset of `S` that is `< U`.
  - `S.tailSet(L)`: subset that is $\geq L$.
  - `S.subSet(L,U)`: subset that is $\geq L$, `< U`.

- **Changes to views modify** `S`.

- **Attempts to**, e.g., add to a `headSet` beyond `U` are disallowed.

- **Can iterate through a view to process a range**:

  ```java
  SortedSet<String> fauna = new TreeSet<String>(Arrays.asList("axolotl", "elk", "dog", "hartebeest", "duck"));
  for (String item : fauna.subSet("bison", "gnu"))
      System.out.printf("%s, ", item);
  
  would print "dog, duck, elk,"
  ```
TreeSet

- **Java library type** TreeSet\(\langle T \rangle\) requires either that \(T\) be Comparable, or that you provide a Comparator, as in:

  ```java
  SortedSet<String> rev_fauna = new TreeSet<String>(Collections.reverseOrder());
  ```

- **Comparator is a type of function object:**

  ```java
  interface Comparator\langle T \rangle { 
      /** Return <0 if LEFT<RIGHT, >0 if LEFT>RIGHT, else 0. */
      int compare(T left, T right);
  }
  ```

  (We’ll deal with what Comparator\(\langle T \text{ extends Compare}lable\langle T\rangle\rangle\) is all about later.)

- **For example, the reverseOrder comparator is defined like this:**

  ```java
  /** A Comparator that gives the reverse of natural order. */
  static <T extends Comparable<T>> Comparator<T> reverseOrder() {
      // Java figures out this lambda expression is a Comparable<T>.
      return (x, y) -> y.compareTo(x);
  }
  ```
Example of Representation: BSTSet

- Same representation for both sets and subsets.
- Pointer to BST, plus bounds (if any).
- `.size()` is expensive!

```java
SortedSet<String> fauna = new BSTSet<String>(stuff);
subset1 = fauna.subSet("bison","gnu");
subset2 = subset1.subSet("axolotl","dog");
```