CS 228: Introduction to Data Structures
Lecture 33
Wednesday, April 15, 2015

Heaps (continued)

Bounding the Height of a Heap

Last time, we claimed that the following holds, and that it implies that `add` and `remove` take O(log n) time in the worst case.

Height Bound. The height of an n-node heap is at most $\log_2 n$.

To see why this bound is true, let h be the height of the heap, and n be the number of nodes. Then, by drawing some pictures, you can convince yourselves that

- if h = 0, then $2^0 \le n = 1 \le 2^{0+1}-1$,
- if h = 1, then $2^1 = 2 \le n \le 3 = 2^{1+1}-1$,
- if h = 2, then $2^2 = 4 \le n \le 7 = 2^{2+1}-1$,
- if h = 3, then $2^3 = 8 \le n \le 15 = 2^{3+1}-1$, etc.

More generally, we can argue that

$$2^h \le n \le 2^{h+1}-1.$$

The left-hand inequality, $2^h \le n$, proves the height bound $h \le \log_2 n$. In fact, the inequalities $2^h \le n$ and $n \le 2^{h+1}-1$ together imply that $h = \text{floor}(\log_2 n)$.

Building a Heap

Suppose we want to make a heap out of a collection of n keys. If we `add` them one by one, constructing the heap takes O(n log n) time. There is a faster algorithm, called `heapify`, which works as follows.

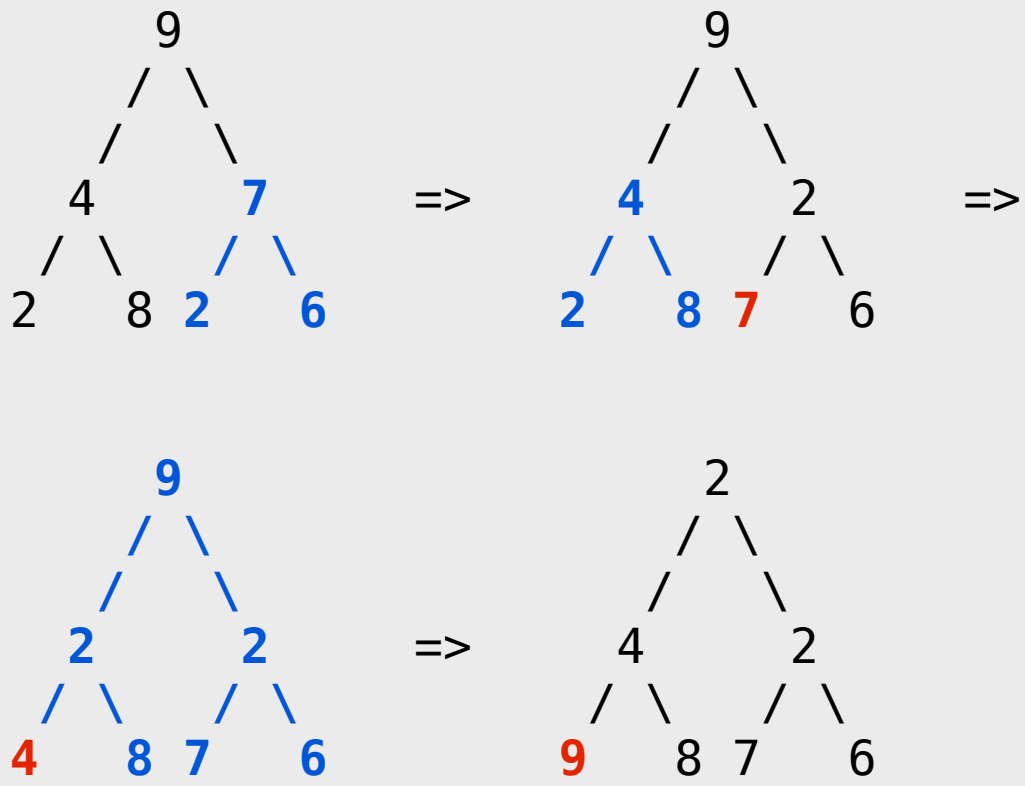
- First, make a complete tree out of the entries by just throwing them, in any order, into an array. (If they're already in the array, just leave them in this order.)

- Then, work backward from the last internal node (non-leaf node) to the root node, in reverse order in the array or the level-order traversal. When we visit a node, percolate its entry down the heap as in `remove`.

To see that this is correct, notice that —since we're going bottom up— before we percolate an entry down, we know (inductively) that its two child subtrees are heaps. Hence,

by percolating the entry down, we create a larger heap rooted at the node where that entry started.

Example.



T**ime complexity.** In the worst case, each internal node percolates all the way down. Thus, the worst-case running time of `heapify` is proportional to the sum of the heights of all the nodes in the tree. It can be shown that this sum is less than n, where n is the number of entries being coalesced into a heap[1]. Hence, the running time is O(n).

[1]Proving this is rather tricky. For a nice "visual" proof, see Goodrich and Tamassia's text, *Data Structures and Algorithms in Java*.

This beats the O(n log n) time it takes to insert n entries into a heap individually.

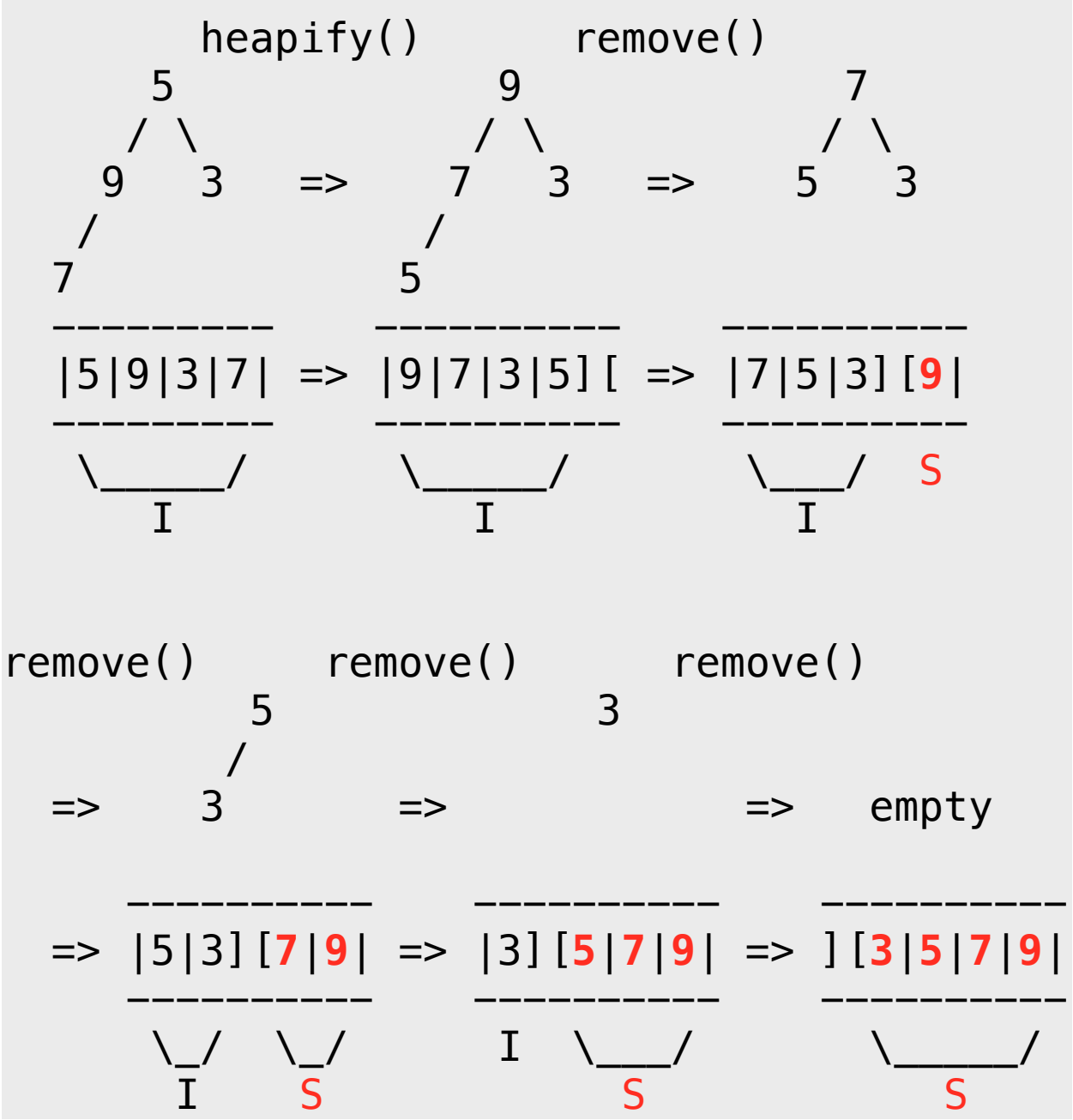
Heapsort

Binary heaps give us another O(n log n) sorting algorithm. We use a ***max heap***; i.e., one where the highest-priority item is the one with the largest key, and, thus, this item is at the root of the heap. A max heap is easily implemented by defining the appropriate comparator (see the code on Blackboard). We sort as follows.

```
HEAPSORT:
|   put the elements into a max heap H
|   create an empty list L
|   while !H.isEmpty()
|   |   x = H.remove()
|   |   Put x at the beginning of L
|   return L
```

In fact, if the input elements are in an array, we can do the sorting ***in place***. That is, we can use the same array to store the heap H and the list L. As items are removed, the heap shrinks toward the beginning of the array, making room for the elements removed.

Example.

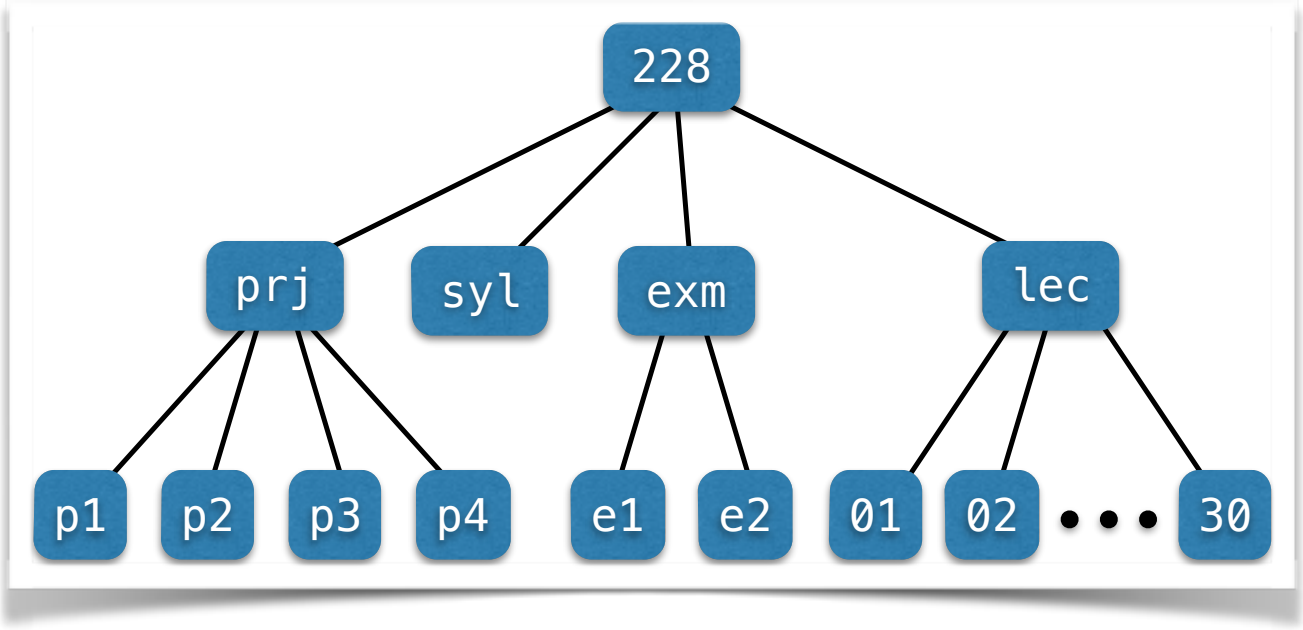


I = in heap

S = sorted

Non-Binary Rooted Trees

All trees we have seen so far — expression trees, BSTs, heaps — are binary. ***Non***-binary trees also have several uses. For instance, file directories are typically tree-structured. Here is an early version of a directory for this class.



More generally, a rooted tree can represent all sorts of hierarchical information; e.g. organizational charts.

Representing Non-Binary Rooted Trees

We can represent a tree using a data structure where every node has one reference to the data stored at that node, one reference to that node's parent, and one reference to a list of that node's children. The latter can be stored, for instance, as a linked list.

The ***child-sibling representation*** is another popular tree representation. As before, each node has `data` and `parent` references; however, instead of referencing a list of children, each node references just its leftmost child. Each node also references its next sibling to the right. These `nextSibling` references are used to join the children of a node in a singly-linked list, whose head is the node's `firstChild`. Here are the basic definitions; the full code is on Blackboard[2].

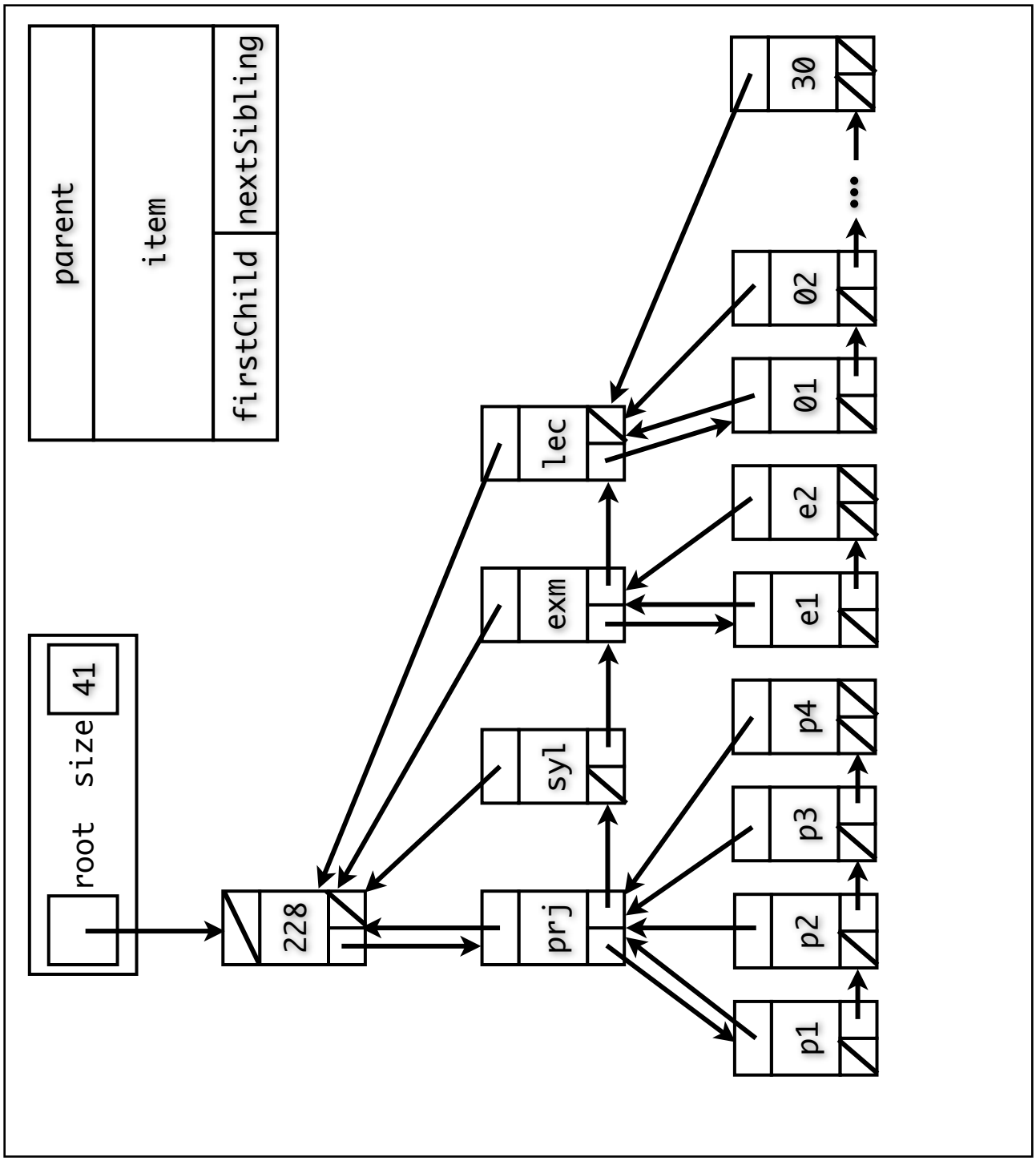
[2] See “Lecture Material > Mar 13”. (Section B covered this material before Spring Break.)

```
public class CSNode<E>
{
  protected CSNode<E> parent;
  protected CSNode<E> firstChild;
  protected CSNode<E> nextSibling;
  protected E data;

  public CSNode() {}

  public CSNode(E data)
  {
    this(data, null, null);
  }

  public CSNode(E data, CSNode<E> child,
                CSNode<E> sibling)
  {
    this.firstChild = child;
    this.nextSibling = sibling;
    this.data = data;
  }
```

A tree is represented by a reference to the root node, and the tree's size (number of nodes).

```
public class CSTree<E> {
  CSNode<E> root;
  int size;
}
```

Here is an example.



Traversing Non-Binary Trees

Preorder, postorder, and level-order traversal easily extend to non-binary trees (inorder traversal does not make sense for non-binary trees). For instance, a preorder traversal is a natural way to print a directory structure:

```
~dfb/228
   prj
      p1
      p2
      p3
      p4
   syllabus
   exm
      e1
      e2
   lec
      01
      02
      .
      .
      .
      30
```

We will see how to traverse non-binary trees next time.

Appendix 1: Pseudocode for Heapify

In the following pseudocode

- `data` is an array that contains the elements of the heap,
- `size` is the number of elements in the heap, and

For simplicity, we assume that the heap consists of integer keys. The Java code posted on Blackboard allows more general objects.

```
heapify(data):

   current = size/2 – 1

   while (current >= 0)
      percolateDown(data, current)      (*)
      --current
```

We claim that the following holds at the beginning of each iteration of the **`while`** loop.

Loop invariant: For each `i > current`, the subtree rooted at `data[i]` is heap-ordered.

Before entering the loop, `current` is set to the index of last internal node, so the invariant holds trivially. At the

beginning of each iteration, each child of `current` has index > `current`. By the loop invariant, both subtrees of `current` are heap-ordered. By the postcondition of `percolateDown()`, the subtree at `current` is heap ordered after line (*). Thus, after decrementing `current`, the loop invariant still holds.

Appendix 2: Other Types of Heaps
(Not Covered in Class and not Examinable)

Binary heaps are not the only heaps in town. ***Mergeable heaps*** are an important class of heaps. Their main characteristic is that two mergeable heaps can be combined together quickly into a single mergeable heap. We will not describe these complicated heaps in CS 228, but it's good to know they exist in case you ever need one.

The best-known mergeable heaps are ***binomial heaps***, ***Fibonacci heaps***, ***skew heaps***, and ***pairing heaps***. Fibonacci heaps have another remarkable property: if you have a reference to an arbitrary node in a Fibonacci heap, you can decrease its key in constant time. (Pairing heaps are suspected of having the same property, but nobody knows for sure.) This operation is used frequently by an important algorithm for finding the shortest path in a graph.

The next table compares the different heaps. The time bounds for skew heaps, pairing heaps, and Fibonacci heaps are ***amortized*** bounds, not worst case bounds. This means that, if you start from an empty heap, any sequence of operations will take no more than the given time bound on average, although individual operations may occasionally take longer. All other running times are worst-case.

	Binary	Binomial	Skew	Pairing	Fibonacci
add()	O(log n)	O(log n)	O(1)	O(log n)*	O(1)
remove()	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
merge()	O(n)	O(log n)	O(1)	O(log n)*	O(1)
decreaseKey()	O(log n)	O(log n)	O(log n)	O(log n)*	O(1)

*Conjectured to be O(1), but nobody has proven or disproven it.