

Outline

- review of Huffman tree implementation (from last lecture, and only if requested)
- review of lambda expressions (from ICS 111)
- heaps
- heap storage in arrays
- heap insertion and removal
- priority queues

an implementation for Huffman trees

data structures

- many possible implementations, this is one (textbook, chapter 6.1 and 6.7)
- use several data structures:
 - a priority queue to hold the sorted data
 - each priority queue element refers to a tree node
 - an interior tree node has no value, but has two children
 - a leaf node has a value, but has no children
 - each node has a frequency of occurrence, which is used as a priority in the queue (low priority returned first)

an implementation for Huffman trees algorithm

- begin by computing the frequency of each value
 - perhaps by using a hash table -- explained later
- once the frequency of values is known, insert each value into the priority queue, using its frequency as a priority
- remove the front two elements from the queue, create an interior node to refer to these two elements, and insert the interior node back into the queue with, as priority, the sum of the priorities of the two nodes
- once there is only one element left in the queue, this is the huffman tree
- a further step would be to build an encoding table from the tree
- in-class exercise: do this given the string "there is no place like home"

lambda expressions

- a lambda expression is an unnamed function
 - that can be assigned to a variable
 - or passed as a parameter
- so another view is that lambda expressions allow function variables
- some languages make heavy use of lambda expressions: for example, Javascript and LISP
- older versions of Java (before Java 8) did not have lambda expressions

lambda expressions in Java

- a lambda expression in Java has 3 parts:
 - the interface definition, defining the type
 - the lambda expression, defining the computation
 - the application of the lambda expression

lambda expressions example

- `Comparator<T> defines int compare(T t1, T t2)`
- `Comparator<T> f = (a, b) -> {
 if (a.compareTo(b) > 0) return 1;
 if (a.compareTo(b) == 0) return 0;
 return -1;
}`
- `int comparison=f(value, a[middle]);`
- **note:** `f` doesn't have to be called `compare`

Defining the interface

- any interface that declares exactly one method is a functional interface, and can be used to declare a lambda expression
- common examples include:
 - `Comparator<T>` defines `int compare(T a, T b)`
 - `Function<T, R>` defines `R apply(T t)`
 - `Predicate<T>` defines `boolean test (T t)`
- anyone can use these types
- or you may define your own

Defining the computation

- a lambda expression has two parts, joined by the arrow: \rightarrow
- the first part is the parameter list
 - types may be omitted if Java can figure them out
 - parentheses may be omitted if only one parameter
 - or use () to indicate no parameters
- the second part is the expression, enclosed in {}
 - the parameter values may be used in the expression
 - the expression may also use local variables of the enclosing method – this is called a *closure*

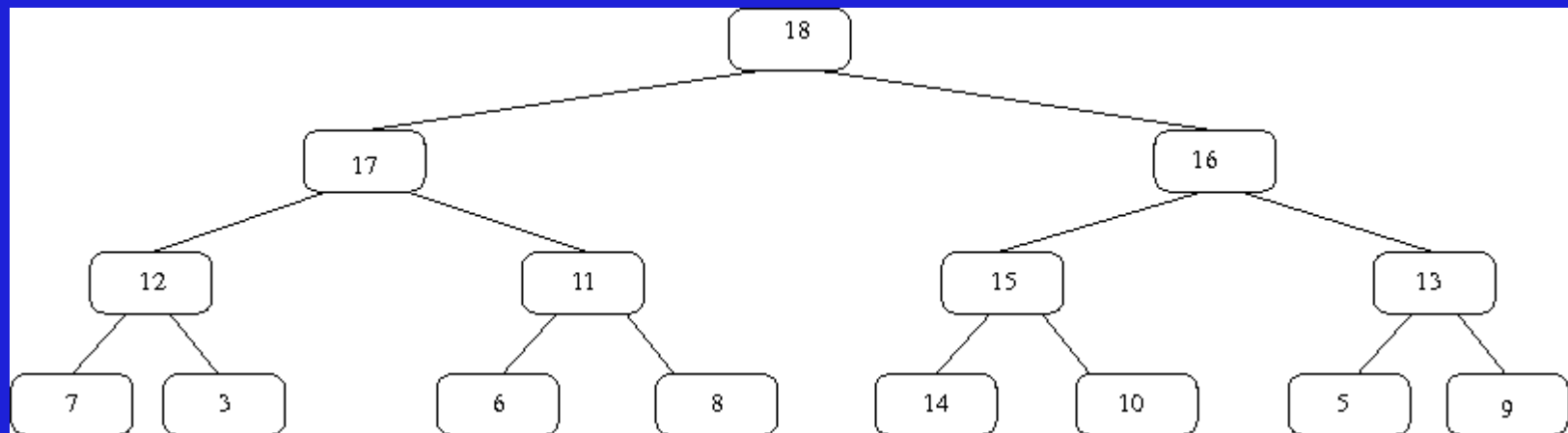
applying lambda expressions

- a lambda expression defined by interface I
- simply call the only method of I
- example:
- ```
Function<String, Integer> f =
 s -> { return s.length(); } ;
```
- ```
Integer x = f.apply("hello world")  
+ 2;
```

Heaps

- a **heap** is a binary tree
 - in which each node has a value less than its children (min heap)
 - or a value greater than its children (max heap)
- this is the **heap property**
- unlike a binary search tree, nodes in a heap are **not** sorted overall
 - instead, the heap property insures that the largest or smallest value is at the top of the heap

heap example



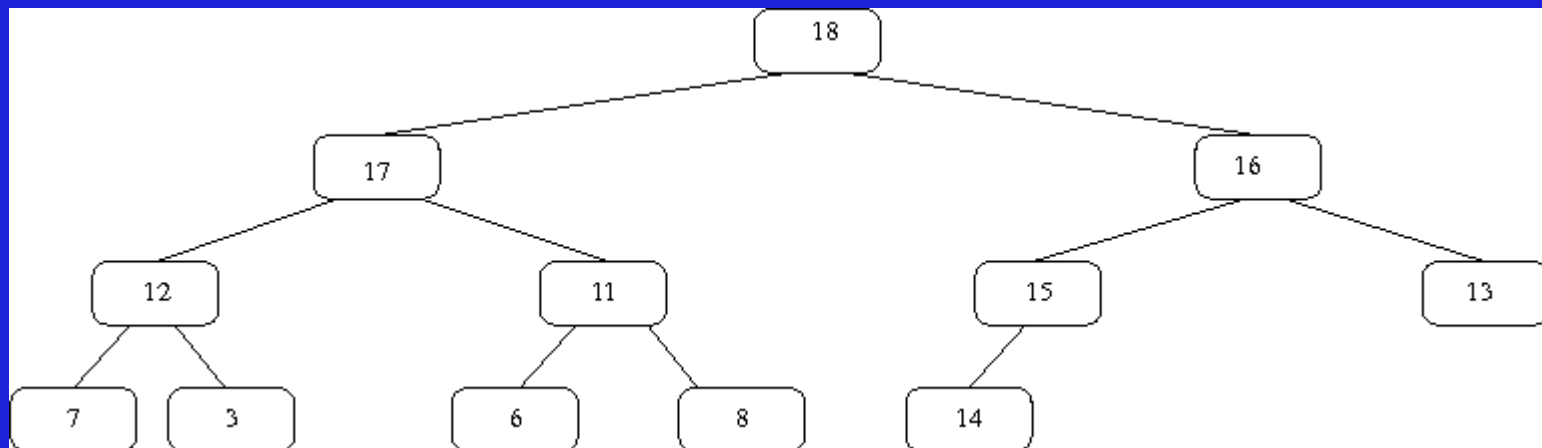
- is this a min heap or a max heap?

Heap Requirements

- as well as the heap property of each node needing to be less (greater) than its children, in a heap, all leaves at maximum depth d_{\max} are as far to the left as possible.
- any other leaves are at depth $d_{\max} - 1$
- such a binary tree is called a **complete** binary tree

Heap Requirements

- a heap is always a complete binary tree
- a complete binary tree is always balanced, so that a complete binary tree of n nodes has depth $O(\log n)$
- in-class exercise: in what sense can I say that a complete binary tree is balanced?



Heap Storage

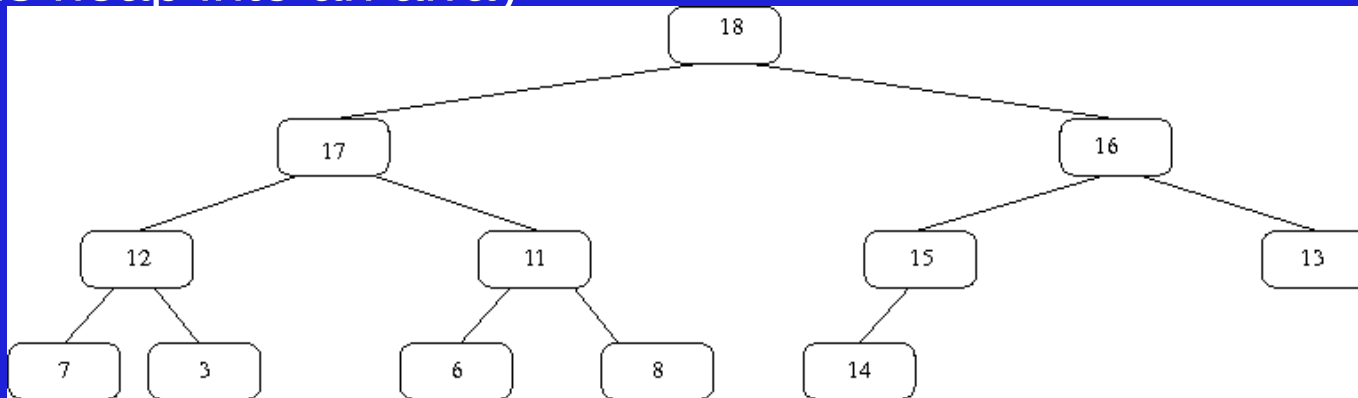
- any complete binary tree, including any heap, can be conveniently stored in an array:
 - element 0 of the array stores the root
 - elements 1 and 2 of the array store the nodes at depth 2
 - elements 3, 4, 5, and 6 of the array store the nodes at depth 3
 - nodes at depth d are stored in array elements $2^{d-1}-1 \dots 2^d - 2$
- conveniently, there is no need to store any pointers!

Array storage (without pointers) of a Complete Binary Tree

- a node stored in array element i has:
 - its parent in array element $(i - 1) / 2$
 - its left child (if any) in array element $2i + 1$
 - its right child (if any) in array element $2i + 2$
- since a heap is a complete binary tree, the right child can only be present if the left child is also present
 - there may be one left child leaf node that does not have a right sibling

in-class exercises

- store this heap into an array



- which of the following arrays store max heaps, min heaps, or neither?

index	0	1	2	3	4	5	6	7	8	9
array 1	0	1	2	0	4	5	6	7	8	9
array 2	9	8	7	6	5	4	3	2	1	0
array 3	5	5	5	6	6	6	6	7	7	1
array 4	9	3	9	2	1	6	7	1	2	1
array 5	8	7	6	1	2	3	4	2	1	2

Heap Insertion

- the two heap requirements must be maintained when adding to a heap
- to maintain the complete binary tree property, the new node must be added to the right of all nodes at depth d^{\max}
- or, if there already are $2^{(d^{\max})}$ nodes at that level, the new node should be inserted all the way to the left, making the tree deeper by one level
- either way, the new value is inserted **in the array** just after all elements already in the array
- now the tree is complete, but may not have the heap property
- to check, compare the new node with the parent, and swap the two if needed to maintain the heap property
- continue with the parent's parent, all the way to the root if necessary
- now the complete binary tree also obeys the heap property

Heap Deletion

- the largest node is at the root of the heap
- that node is removed, and replaced with the bottommost, rightmost node (the node at the end of the array)
- the remaining tree is complete, but may not have the heap property
- if the root node is less than either of its children, it is swapped with the largest of its children
- the operation continues with the new node
- now the complete binary tree also obeys the heap property
- similarly for min heaps

priority queues

- the queues studied so far were strictly FIFO
- that means objects were returned in the order inserted
- in the real world, often have priorities, e.g.:
 - at airport check-in, there is a special line for first-class passengers
 - if any first-class passengers are waiting, they are handled first
 - if no first-class passengers are waiting, only then the other passengers can check-in
- likewise some kinds of traffic in a network get priority
 - e.g. identified real-time traffic on WiFi

priority queue implementations

- linked list: objects are inserted in the proper place in the list (linear time), objects are returned from the front of the list (constant time)
 - nice because insertion of high priority items is fast
- array: objects are inserted in the proper place in the array, with all other objects shifted to make room (linear time), objects are returned from the front of the array, with all other objects shifted down (linear time), or maintained as a circular array (constant time)
- binary search tree: objects are inserted into the binary search tree, using the priority as the key (log or linear), objects are removed from the leftmost or rightmost node of the tree (log or linear)
 - unless a balanced tree algorithm is used, the tree tends to become unbalanced, because nodes are always removed from the same side, so time becomes linear
- heap: objects are inserted into the heap using the priority as the key (log time), and removed from the top of the heap (log time)
 - the simplest algorithm with guaranteed log time operations!

priority queue performance

- if the priority queue only has a few elements, any of these implementations is fine
- however, if the priority queue might grow long, then frequent operations should be fast
- the performance depends on the algorithm, on the operation, and on the priority
- e.g., always adding something with highest priority is fast if using a linked list