Reproduction rights obtainable from www.CartoonStock.com

"You're reordering your priorities? Since when do you have *priorities?*"

CS202 Fall 2012 Lecture

Priority Queue

Prof. Tanya Berger-Wolf

Motivating examples

we are a bank and we want to manage a collection of our customers' information

- we'll be performing many adds and removes
- we'd like to be able to ask the collection to `remove` or `get` the information about the customer with the minimum bank account balance

another example: A shared Unix server has a list of print jobs to print. It wants to print them in chronological order, but each print job also has a *priority*, and higher-priority jobs always print before lower-priority jobs

another example: We are writing a parking search algorithm. It needs to search for the best parking spot near a location; it will enqueue all possible paths to potential open parking spots with priorities (based on various parameters), and try them in order

2

Some bad implementations

list: store all customers/jobs in an unordered list, remove min/max one by searching for it

- problem: expensive to search

sorted list: store all in a sorted list, then search it in O(log *n*) time with binary search

- problem: expensive to add/remove

binary search tree: store in a BST, search it in O(log *n*) time for the min (leftmost) element

- problem: tree could be unbalanced on nonrandom input

balanced BST

- problem: in practice, if the program has many adds/removes, it performs slowly on AVL trees and other balanced BSTs
- problem: removal always occurs from the left side, which unbalances the ree

3

Priority queue ADT (Weiss 21.1

priority queue: an collection of ordered elements that provides fast access to the minimum (or maximum) element

- a mix between a queue and a BST
- usually implemented using a structure called a *heap*

priority queue operations:

- `add`
  - *O*(1) average, *O*(log *n*) worst-case
- `peek` - returns the **minimum** element
  - *O*(1)
- `removeMin` - removes/returns **minimum** element
  - *O*(log *n*) worst-case
- `isEmpty`, `clear`, `size`, `iterator`
  - *O*(1)

4

Java's PriorityQueue class

```
public class PriorityQueue<E> implements Queue<E>
public void clear()
public Iterator<E> iterator()
public boolean offer(E o)
public E peek()
public E poll()
public E remove()
```

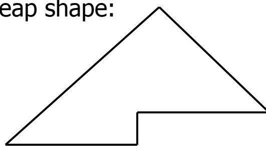

Implementing a priority queue using a heap

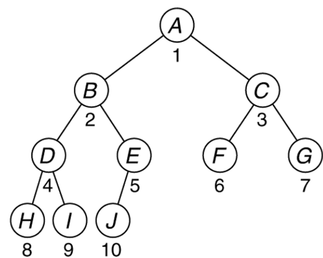
Heap properties (Weiss 21.1.)

heap: a tree with the following two properties:

- 1. ***completeness*** (Weiss 6.3.1)
  complete tree: every level is full except possibly the lowest level, which must be filled from left to right with no leaves to the right of a missing node (i.e., a node may not have any children until all of its possible siblings exist)

Heap shape:

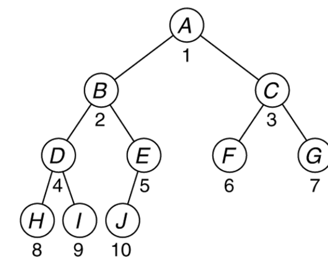




Heap properties 2

- 2. ***heap ordering*** (Weiss 6.3.2)
  a tree has *heap ordering* if $P < X$ for every element X with parent P
  - in other words, in heaps, parents' element values are always smaller than those of their children
  - implies that minimum element is always the root
  - is every a heap a BST? Are any heaps BSTs? Are any BSTs heaps?

$P \leq X$



Which are min-heaps?

A wrong!

B wrong!

C wrong!

D

E

F wrong!

Which are max-heaps?

A

B

C wrong!

D

E wrong!

F

Heap height and runtime

height of a complete tree is always **log *n***, because it is always balanced

- this suggests that **searches**, **adds**, **and removes** will have **O(log *n*)** worst-case runtime
- because of this, if we implement a priority queue using a heap, we can provide the O(log *n*) runtime required for the `add` and `remove` operations

n-node complete tree of height h:

$h = \lfloor \log n \rfloor$

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

Adding to a heap (Weiss 21.2)

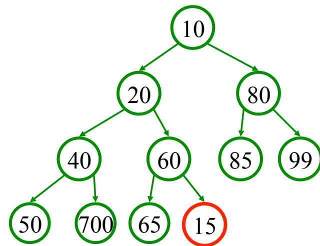
when an element is added to a heap, it should be initially placed as the rightmost leaf (to maintain the completeness property)

- heap ordering property becomes broken!

Adding to a heap, cont'd.

to restore heap ordering property, the newly added element must be shifted upward ("bubbled up") until it reaches its proper place

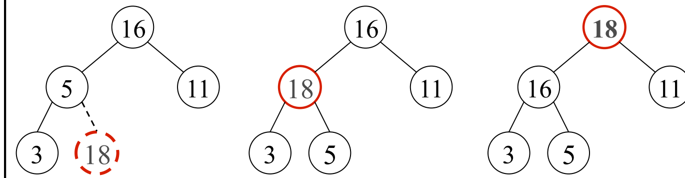
- bubble up (book: "percolate up") by swapping with parent
- how many bubble-ups could be necessary, at most?



13

Adding to a max-heap

same operations, but must bubble up *larger* values to top



14

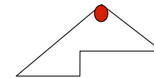
Heap practice problem

Draw the state of the heap tree after adding the following elements to it:

6, 50, 11, 25, 42, 20, 104, 76, 19, 55, 88

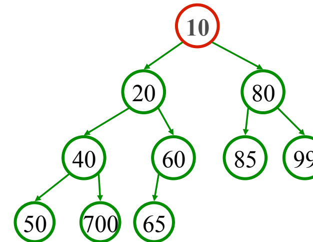
15

The `peek` operation



`peek` on a min-heap is trivial; because of the heap properties, the minimum element is always the root

- `peek` is *O*(1)
- `peek` on a max-heap would be *O*(1) as well, but would return you the maximum element and not the minimum one



16

Removing from a min-heap

min-heaps only support `remove` of the min element (the root)

- must remove the root while maintaining heap completeness and ordering properties
- intuitively, the last leaf must disappear to keep it a heap
- initially, just swap root with last leaf (we'll fix it)

Removing from heap, cont'd.

must fix heap-ordering property; root is out of order

- shift the root downward ("bubble down") until it's in place
- swap it with its smaller child each time

Turning any input into a heap (Weiss 21.3)

we can quickly turn any complete tree of comparable elements into a heap with a `buildHeap` algorithm

simply perform a "bubble down" operation on every node that is not a leaf, starting from the rightmost internal node and working back to the root

- why does this `buildHeap` operation work?
- how long does it take to finish? (big-Oh)

Array tree implementation

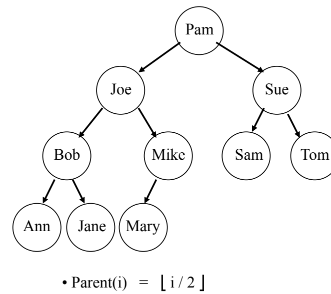
corollary: any complete binary tree can be implemented using an array (the example tree shown is *not* a heap)

i	Item	Left Child	2*i	Right Child
1	Pam	2	2	3
2	Joe	4	4	5
3	Sue	6	6	7
4	Bob	8	8	9
5	Mike	10	10	-1
6	Sam	-1	11	-1
7	Tom	-1	13	-1
8	Ann	-1	15	-1
9	Jane	-1	17	-1
10	Mary	-1	19	-1

- LeftChild(i) = 2*i
- RightChild(i) = 2*i + 1

Array binary tree - parent

i	Item	Parent	i / 2
1	Pam	-1	0
2	Joe	1	1
3	Sue	1	1
4	Bob	2	2
5	Mike	2	2
6	Sam	3	3
7	Tom	3	3
8	Ann	4	4
9	Jane	4	4
10	Mary	5	5

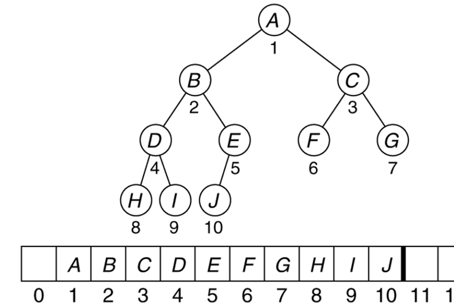


• Parent(i) = $\lfloor i / 2 \rfloor$

Implementation of a heap

when implementing a complete binary tree, we actually can "cheat" and just use an array

- index of root = 1 (leave 0 empty for simplicity)
- for any node *n* at index *i*,
  - index of *n*.left = 2*i*
  - index of *n*.right = 2*i* + 1



Code for add (insert) method

```
public void insert(AnyType x) {
    // grow array if needed
    if (currentSize >= array.length - 1)
        enlargeArray(array.length*2 + 1);

    // place element into heap at bottom
      int hole = ++currentSize;
      for ( ;
            hole > 1 && x.compareTo(array[hole/2]) < 0;
            hole /=2 )
        array[hole] = array[hole/2]
}
```


Code for peek (findMin) method

```
public AnyType peek() {
    if (isEmpty()) {
        throw new UnderflowException();
    }

    return array[1];
}
```

Code for remove (deleteMin) method

```
public AnyType remove() {
    AnyType result = peek();

    // move last element of array up to root
    array[1] = array[currentSize--];

    percolateDown( 1 );
    return result;
}
```


The bubbleUp helper

```
private void bubbleUp() {
    int index = this.size;

    while (hasParent(index) &&
        elements[index].compareTo(parent(index)) < 0) {

        // parent/child are out of order; swap them
        int parentIndex = parentIndex(index);
        swap(elements, index, parentIndex);
        index = parentIndex;
    }
}

// helpers
private boolean hasParent(int i) { return i > 1; }
private int parentIndex(int i)   { return i/2; }
private E parent(int i)  { return elements[parentIndex(i)]; }
```


The percolateDown helper

```
private void percolateDown(in hole) {
    int index = 1;
    while (hasLeftChild(index)) {
        int childIndex = leftIndex(index);
        if (hasRightChild(index) &&
            right(index).compareTo(left(index)) < 0) {
            childIndex = rightIndex(index);
        }

        if (elements[index].compareTo(elements[childIndex]) > 0) {
            swap(elements, index, childIndex);  // out of order
            index = childIndex;
        } else {
            break;
        }
    }
}

// helpers
private int leftIndex(int i)   { return i*2; }
private int rightIndex(int i)  { return i*2 + 1; }
private boolean hasLeftChild(int i)  { return leftIndex(i) <= size(); }
private boolean hasRightChild(int i) { return rightIndex(i) <= size(); }
```


The percolateDown helper

```
private void percolateDown(int hole) {
    in child;
    AnyType tmp = array[hole];

    for( ; hole*2 <= currentSize; hole = child) {
        child = hole * 2
        if (child != currentSize &&                  // smaller child
                array[child + 1].compareTo( array[child]) < 0)
            child++;
        if (array[child].compareTo( tmp ) < 0) //parent greater than child
            array[hole] = array[child]         //swap
        else
            break;
    }
    array[hole] = tmp;
}
```

Advantages of array heap

the "implicit representation" of a heap in an array makes several operations very fast

- add a new node at the end (O(1))
- from a node, find its parent (O(1))
- swap parent and child (O(1))
- a lot of dynamic memory allocation of tree nodes is avoided
- the algorithms shown usually have elegant solutions

```
private void buildHeap() {
    for (int i = array.currentSize / 2; i > 0;
i--) {
        bubbleDown(i);
    }
}
```

29

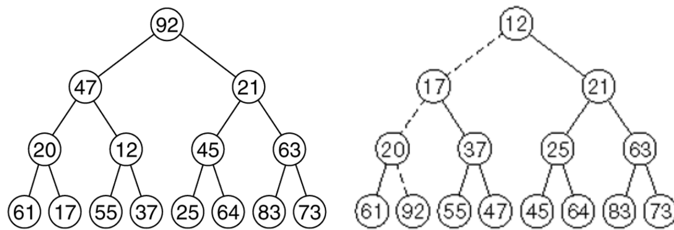
Heap sort
Weiss 21.5

30

Heap sort

heap sort: an algorithm to sort an array of N elements by turning the array into a heap, then doing a `remove` N times

- the elements will come out in sorted order!
- we can put them into a new sorted array
- what is the runtime?

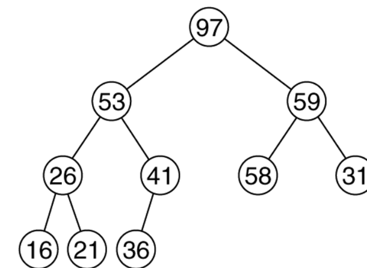


31

A max-heap

the heaps shown have been minimum heaps because the elements come out in ascending order

a *max-heap* is the same thing, but with the elements in descending order



32

Improved heap sort

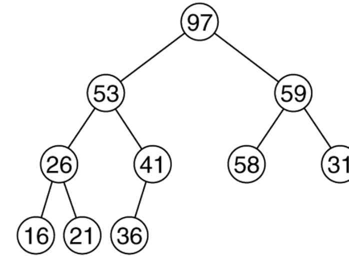
the heap sort shown requires a second array

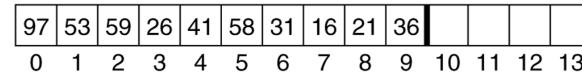
we can use a max-heap to implement an improved version of heap sort that needs **no** extra storage

- $O(n \log n)$ runtime
- no external storage required!
- useful on low-memory devices
- elegant

Improved heap sort 1

use an array heap, but with 0 as the root index
max-heap state after `buildHeap` operation:

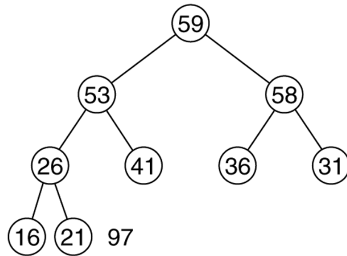


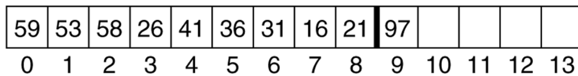


Improved heap sort 2

state after one `remove` operation:

- modified `remove` that moves element to end

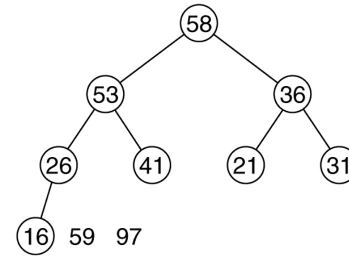


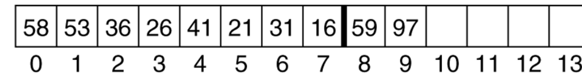


Improved heap sort 3

state after two `remove` operations:

- notice that the largest elements are at the end (becoming sorted!)





Sorting algorithms review

	Best case	*Average* case (†)	*Worst* case
Selection sort	n^2	n^2	n^2
Bubble sort	n	n^2	n^2
Insertion sort	n	n^2	n^2
Mergesort	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
Heapsort	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$
Treesort	$n \log_2 n$	$n \log_2 n$	n^2
Quicksort	$n \log_2 n$	$n \log_2 n$	n^2

† According to Knuth, the **average growth rate** of Insertion sort is about 0.9 times that of Selection sort and about 0.4 times that of Bubble Sort. Also, the **average growth rate** of Quicksort is about 0.74 times that of Mergesort and about 0.5 times that of Heapsort.

37