CSE 326: Data Structures
Lecture #3
Mind your Priority Qs

Bart Niswonger

Summer Quarter 2001

Today's Outline

- The First Quiz!
- Things Bart Didn't Get to on Wednesday (Priority Queues & Heaps)
- Extra heap operations
- d-Heaps

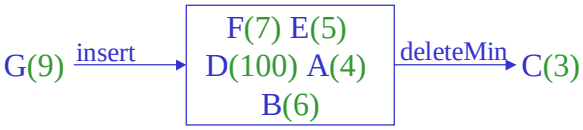
Back to Queues

- Some applications
  - ordering CPU jobs
  - simulating events
  - picking the next search site
- Problems?
  - short jobs **should go first**
  - earliest (simulated time) events **should go first**
  - most promising sites **should be searched first**

Remember ADTs?

Priority Queue ADT

- Priority Queue operations
  - create
  - destroy
  - insert
  - deleteMin
  - is_empty



- Priority Queue property: for two elements in the queue, *x* and *y*, if *x* has a lower priority value than *y*, *x* will be deleted before *y*

Applications of the Priority Q

- Hold jobs for a printer in order of length
- Store packets on network routers in order of urgency
- Simulate events
- Select symbols for compression
- Sort numbers
- Anything *greedy*

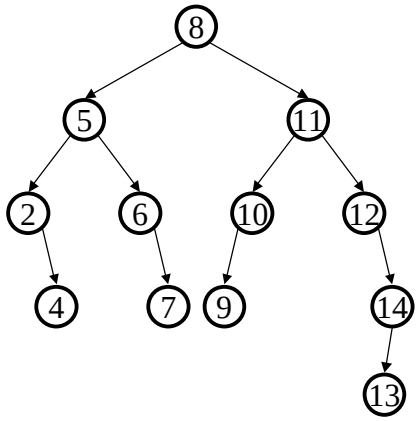
Naïve Priority Q Data Structures

- Unsorted list:
  - *insert:*
  - *deleteMin:*
- Sorted list:
  - *insert:*
  - *deleteMin:*

Binary Search Tree Priority Q Data Structure (that's a mouthful)

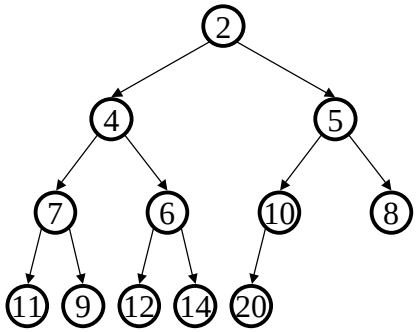
insert:

deleteMin:



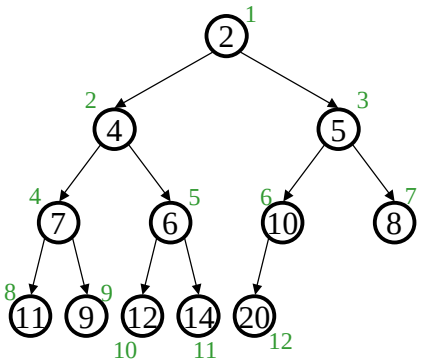
Binary Heap Priority Q Data Structure

- Heap-order property
  - parent's key is less than children's keys
  - result: minimum is always at the top
- Structure property
  - complete tree with fringe nodes packed to the left
  - result: depth is always O(log n); next open location always known



Nifty Storage Trick

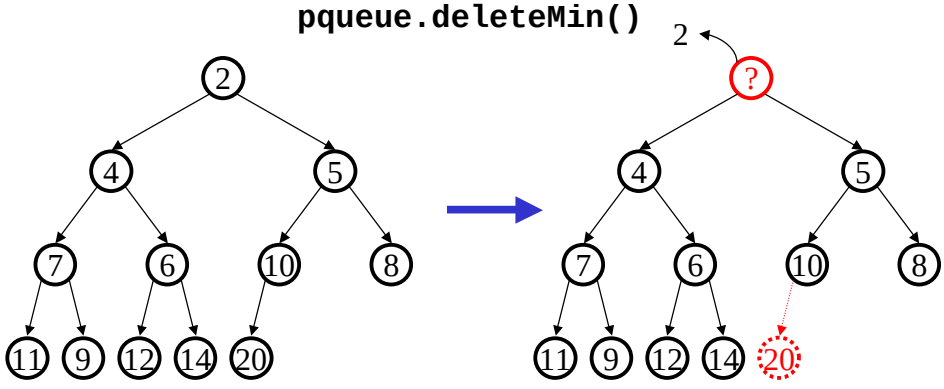
- Calculations:
  - child:
  - parent:
  - root:
  - next free:



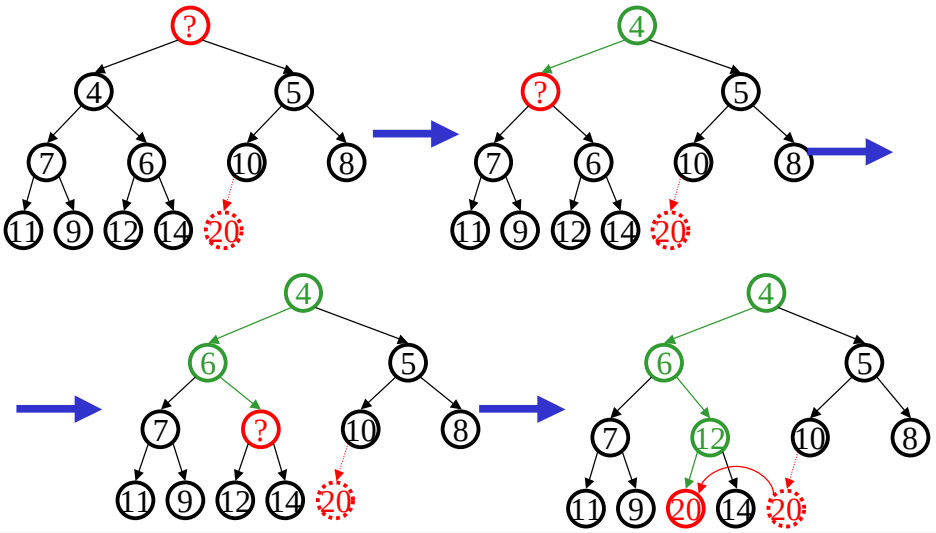
0	1	2	3	4	5	6	7	8	9	10	11	12	
12	2	4	5	7	6	10	8	11	9	12	14	20	

DeleteMin

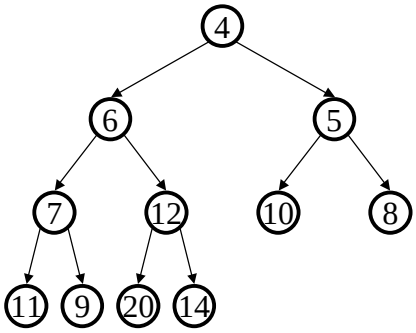
`pqueue.deleteMin()`



Percolate Down



Finally…



DeleteMin Code

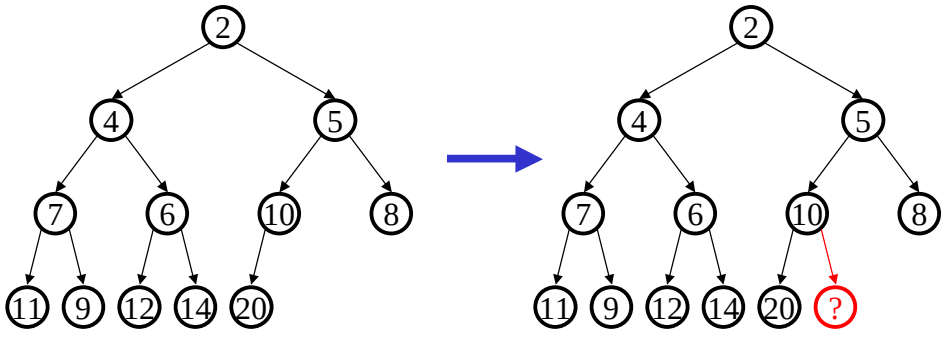
```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[1];
  size--;
  newPos =
    percolateDown(1,
        Heap[size+1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

runtime:

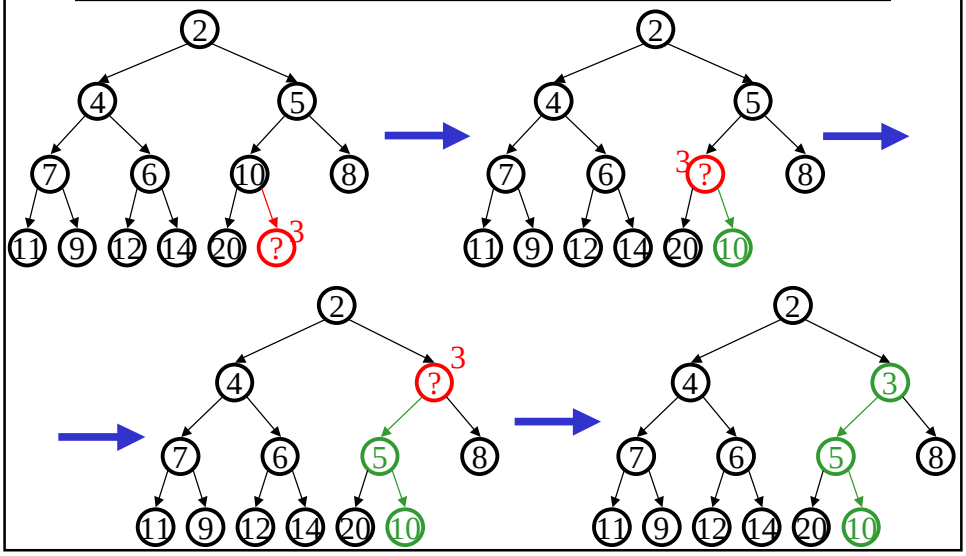
```
int percolateDown(int hole, Object val) {
  while ( 2 * hole <= size ) {
    left = 2 * hole;
    right = left + 1;
    if ( right <= size &&
         Heap[right] < Heap[left])
      target = right;
    else
      target = left;

    if ( Heap[target] < val ) {
      Heap[hole] = Heap[target];
      hole = target;
    }
    else
      break;
  }
  return hole;
}
```

Insert

pqueue.insert(3)



Percolate Up



Insert Code

```
void insert(Object o) {
  assert(!isFull());
  size++;
  newPos =
    percolateUp(size,o);
  Heap[newPos] = o;
}
```

```
int percolateUp(int hole,
                Object val) {
  while (hole > 1 &&
         val < Heap[hole/2])
    Heap[hole] = Heap[hole/2];
    hole /= 2;
  }
  return hole;
}
```

runtime:

Changing Priorities

- In many applications the priority of an object in a priority queue may change over time
  - if a job has been sitting in the printer queue for a long time increase its priority
  - unix “renice”
- Must have some (separate) way of find the position in the queue of the object to change (*e.g.* a hash table)

Other Priority Queue Operations

- decreaseKey
  - given the position of an object in the queue, reduce its priority value
- increaseKey
  - given the position of an an object in the queue, increase its priority value
- remove
  - given the position of an object in the queue, remove it
- buildHeap
  - given a set of items, build a heap

DecreaseKey, IncreaseKey, and Remove

```
void decreaseKey(int obj) {
  assert(size >= obj);
  temp = Heap[obj];
  newPos = percolateUp(obj, temp);
  Heap[newPos] = temp;
}

void increaseKey(int obj) {
  assert(size >= obj);
  temp = Heap[obj];
  newPos = percolateDown(obj, temp);
  Heap[newPos] = temp;
}
```

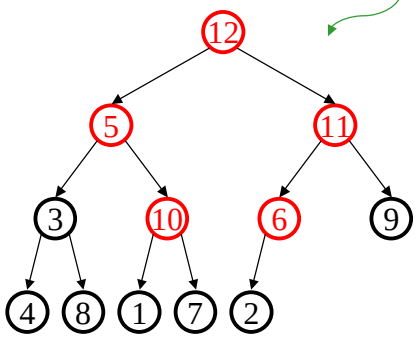
```
void remove(int obj) {
  assert(size >= obj);
  percolateUp(obj,
          NEG_INF_VAL);
  deleteMin();
}
```

BuildHeap

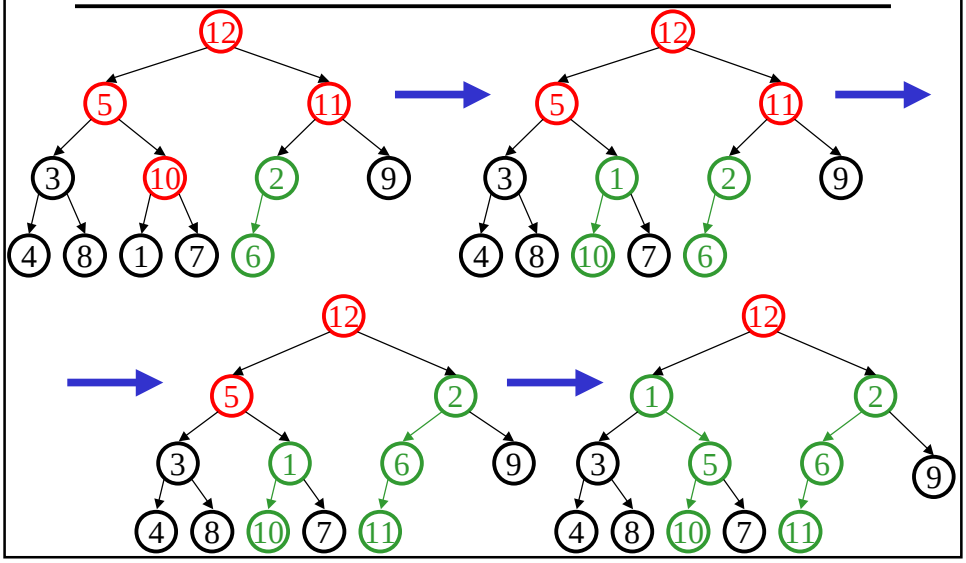
Floyd's Method. Thank you, Floyd.

12	5	11	3	10	6	9	4	8	1	7	2

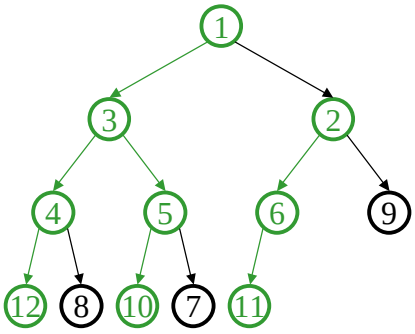
pretend it's a heap and fix the heap-order property!



Build(this)Heap



Finally…



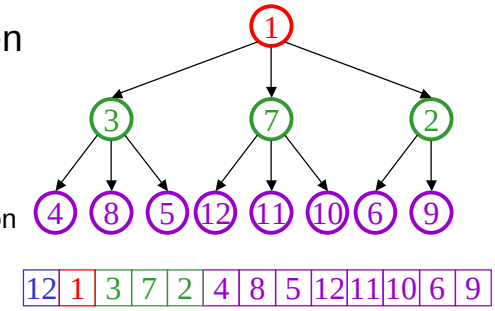
runtime:

Thinking about Heaps

- Observations
  - finding a child/parent index is a multiply/divide by two
  - operations jump widely through the heap
  - each operation looks at only two new nodes
  - inserts are at least as common as deleteMins
- Realities
  - division and multiplication by powers of two are **fast**
  - looking at one new piece of data sucks in a cache line
  - with **huge** data sets, disk accesses dominate

Solution: d-Heaps

- Each node has *d* children
- Still representable by array
- Good choices for *d*:
  - optimize performance based on # of inserts/removes
  - choose a power of two for efficiency
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block



One More Operation

- Merge two heaps. Ideas?

To Do

- Read chapter 6 in the book
- Have teams
- Do project 1
- Ask questions!

Coming Up

- Mergeable heaps
- Dictionary ADT and Self-Balancing Trees
- Unix Development Tutorial (Tuesday)
- First project due (next Wednesday)