# CSE 326: Data Structures
# Lecture #13
# Priority Queues and Binary Heaps

Nick Deibel

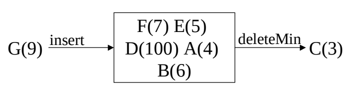
Winter Quarter 2002

## Not Quite Queues

- Consider applications
  - ordering CPU jobs
  - searching for the exit in a maze
  - emergency room admission processing
- Problems?
  - short jobs **should go first**
  - most promising nodes **should be searched first**
  - most urgent cases **should go first**

## 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
- Sort numbers
- Simulate events
- Anything *greedy*

## Discrete Event Simulation

- An event is a pair (x,t) where x describes the event and t is time it should occur
- A discrete event simulator (DES) maintains a set S of events which it intends to simulate in time order

repeat {

Find and remove $(x_0,t_0)$ from S such that $t_0$ minimal;

Do whatever $x_0$ says to do, in the process new events $(x_2,t_2)\ldots(x_k,t_k)$ may be generated;

Insert the new events into S; }

## Emergency Room Simulation

- *Two* priority queues: time and criticality
- *K* doctors work in an emergency room
- events:
  - patients arrive with injury of criticality *C* at time *t* (according to some probability distribution)
    - Processing: if no patients waiting and a free doctor, assign them to doctor and create a future departure event; else put patient in the Criticality priority queue
  - patient departs
    - If someone in Criticality queue, pull out most critical and assign to doctor
- How long will a patient have to wait? Will people die?

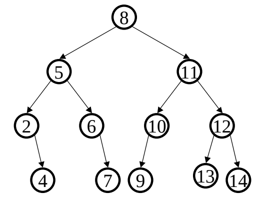
## Naïve Priority Queue Data Structures

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

## BST Tree Priority Queue Data Structure

•Regular BST:
–*insert:*
–*deleteMin:*

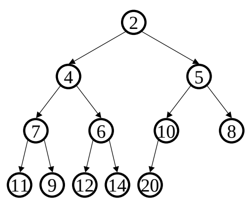
•AVL Tree:
–*insert:*
–*deleteMin:*



*Can we do better?*

## Binary Heap Priority Q Data Structure

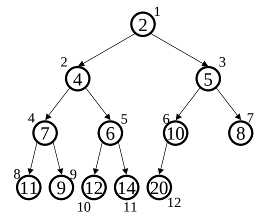
- Heap-order property
  - parent's key is less than children's keys
  - result: minimum is always at the top
- Structure property
  - complete tree with fringe nodes packed to the left
  - result: depth is always O(log n); next open location always known



How do we find the minimum?

## Nifty Storage Trick

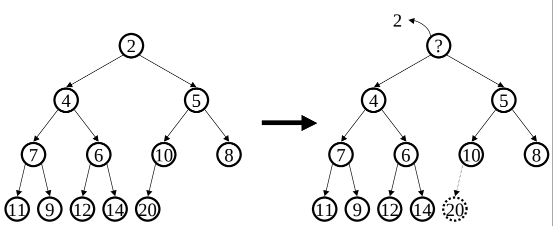
- Calculations:
  - child:
  - parent:
  - root:
  - next free:



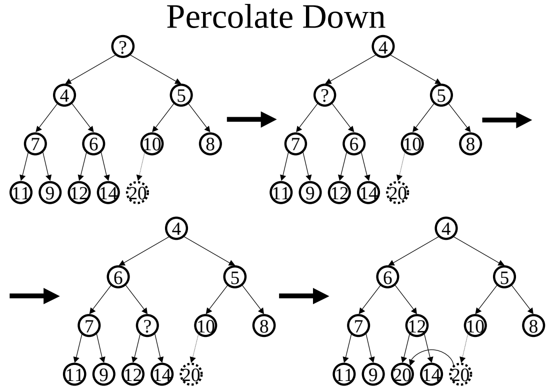
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 2 | 4 | 5 | 7 | 6 | 10 | 8 | 11 | 9 | 12 | 14 | 20 | |

## DeleteMin

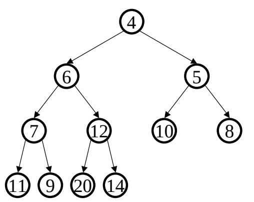
**`pqueue.deleteMin()`**



## Percolate Down



## Finally…



## DeleteMin Code

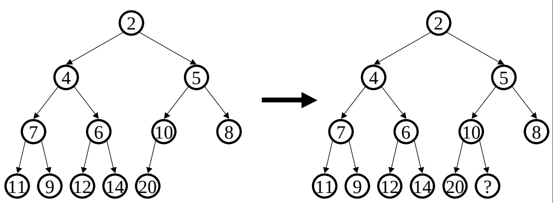
```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[1];
  size--;
  newPos =
    percolateDown(1,
        Heap[size+1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

*runtime:*

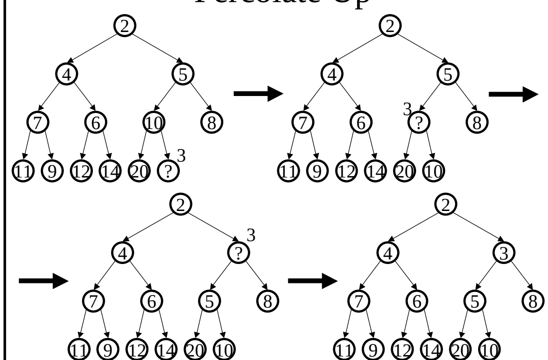
```
int percolateDown(int hole,
                  Object val) {
while (2*hole <= size) {
    left = 2*hole;
    right = left + 1;
    if (right <= size &&
        Heap[right] < Heap[left])
      target = right;
    else
      target = left;

    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:*

## Performance of Binary Heap

| | Binary heap worst case | Binary heap avg case | AVL tree worst case | AVL tree avg case |
|---|---|---|---|---|
| Insert | O(log n) | O(1) percolates 1.6 levels | O(log n) | O(log n) |
| Delete Min | O(log n) | O(log n) | O(log n) | O(log n) |

- In practice: binary heaps much simpler to code, lower constant factor overhead

## 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 an object in the queue, remove it
- buildHeap
  - given a set of items, build a heap

## DecreaseKey, IncreaseKey, and Remove

```
void decreaseKey(int pos, int delta){
  temp = Heap[pos] - delta;
  newPos = percolateUp(pos, temp);
  Heap[newPos] = temp;
}

void increaseKey(int pos, int delta) {
  temp = Heap[pos] + delta;
  newPos = percolateDown(pos, temp);
  Heap[newPos] = temp;
}
```

```
void remove(int pos) {
  percolateUp(pos,
              NEG_INF_VAL);
  deleteMin();
}
```

## BuildHeap

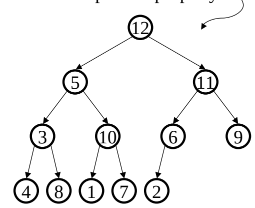
Floyd’s Method. Thank you, Floyd.

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

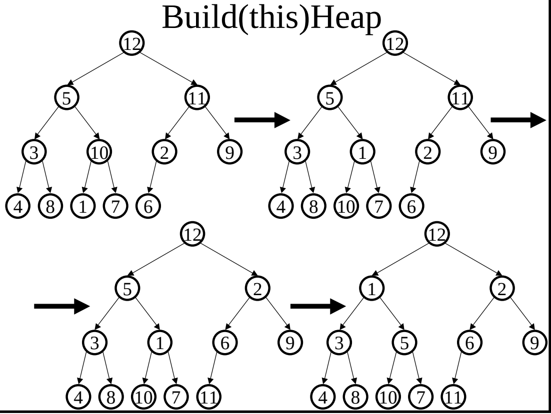
pretend it’s a heap and fix the heap-order property!

Easy worst case bound:

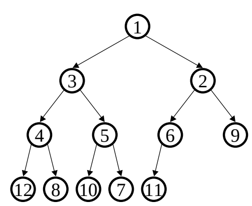
Easy average case bound:




## Build(this)Heap



## Finally…




*runtime?*

## Complexity of Build Heap

- Note: size of a perfect binary tree doubles (+1) with each additional layer
- At most n/4 percolate down 1 level
  at most n/8 percolate down 2 levels
  at most n/16 percolate down 3 levels…

$$1\cdot\frac{n}{4}+2\cdot\frac{n}{8}+3\cdot\frac{n}{16}+\cdots=\sum_{i=1}^{\log n} i\cdot\frac{n}{2^{i+1}}$$

$$=\frac{n}{2}\sum_{i=1}^{\log n}\frac{i}{2^{i}}\le\frac{n}{2}(2)=n$$

O(n)

## Proof of Summation

$$S=\sum_{i=1}^{x}\frac{i}{2^{i}}=\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\cdots+\frac{x-1}{2^{x-1}}+\frac{x}{2^{x}}$$

$$2S=1+\frac{2}{2}+\frac{3}{4}+\cdots+\frac{x}{2^{x-1}}$$

$$S=2S-S=1+\left(\frac{2}{2}-\frac{1}{2}\right)+\left(\frac{3}{4}-\frac{2}{4}\right)+\cdots+\left(-\frac{x}{2^{x}}\right)$$

$$S\le 1+\sum_{i=1}^{x-1}\frac{1}{2^{i}}\le 1+1=2$$

## Heap Sort

- Input: unordered array A[1..N]
  1. Build a max heap (largest element is A[1])
  2. For i = 1 to N-1:
     A[N-i+1] = Delete_Max()

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 50 | 22 | 15 | 4 | 40 | 20 | 10 | 35 | 25 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 50 | 40 | 20 | 25 | 35 | 15 | 10 | 22 | 4 | 7 |
| 40 | 35 | 20 | 25 | 7 | 15 | 10 | 22 | 4 | 50 |
| 35 | 25 | 20 | 22 | 7 | 15 | 10 | 4 | 40 | 50 |

## Properties of Heap Sort

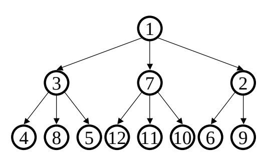
- Worst case time complexity O(n log n)
  - Build_heap O(n)
  - n Delete_Max's for O(n log n)
- In-place sort – only constant storage beyond the array is needed

## 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 terrible 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




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

*What do d-heaps remind us of???*

## Coming Up

- Thursday: Quiz Section is Midterm Review
  - Come with questions!
- Friday: Midterm Exam
  - Bring pencils
- Monday:
  - Mergeable Heaps
  - 3rd Programming project