CSE 326: Data Structures

Topic 3: Priority Queues and Binary Heaps

Ashish Sabharwal
Autumn, 2003

Today's Outline

- Questions on Sound Blaster? (check updates!)
- Finish Asymptotic Analysis
- Trees Review

- **Priority Queues**
- **(Binary) Heaps**
- **d-Heaps**

Simplifying Recurrences

Given a recursive equation for the running time, can sometimes simplify it for analysis.

- For an upper-bound analysis, can optionally simplify to something larger, e.g.

  $T(n) = T(\text{floor}(n/2)) + 1$ to $T(n) \leq T(n/2) + 1$

- For a lower-bound analysis, can optionally simplify to something smaller, e.g.

  $T(n) = 2T(n/2 + 5) + 1$ to $T(n) \geq 2T(n/2) + 1$

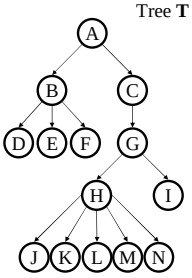
The One Page Cheat Sheet

- **Calculating series:**
  e.g. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
  1. Brute force (Section 1.2.3)
  2. Induction (Section 1.2.5)
  3. Memorize simple ones!
- **Solving recurrences:**
  e.g. $T(n) = T(n/2) + 1$
  1. Expansion (example in class)
  2. Induction (Section 1.2.5, slides)
  3. Telescoping (later…)
- **General proofs (Section 1.2.5)**
  e.g. *How many edges in a tree with n nodes?*
  1. Counterexample
  2. Induction
  3. Contradiction

Tree Review

root(**T**):
leaves(**T**):
children(B):
parent(H):
siblings(E):
ancestors(F):
descendents(G):
subtree(C):

Tree **T**



More Tree Terminology

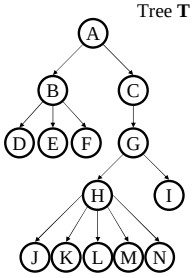
depth(**T**):

height(G):

degree(B):

branching factor(**T**):

Tree **T**

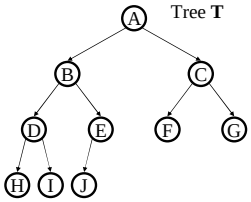


Some More Tree Terminology

T is *binary* if …

T is *n-ary* if …

T is *complete* if …



How deep is a complete tree with *n* nodes?

7

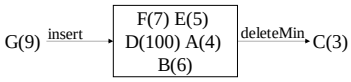
Back to Queues

- 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**

8

A New ADT!

Let's create a new ADT to solve this problem!



What do we need to define this ADT?

9

Priority Queue ADT

1. PQueue data : collection of data with priority
2. PQueue operations
   - create
   - destroy
   - insert
   - deleteMin
   - is_empty
3. PQueue 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*

Note: Often represented as collection of priorities, with data implicit

10

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 with explicit priorities
- Select most frequent symbols for compression
- Sort numbers, picking minimum first
- **Anything *greedy***

11

Naïve Priority Q Data Structures

- Unsorted array:
  - *insert:*
  - *deleteMin:*
- Sorted array:
  - *insert:*
  - *deleteMin:*

Of the two, which is likely to be better?

12

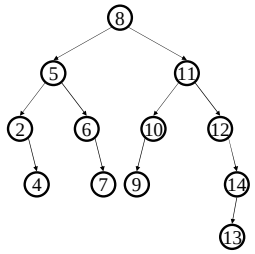
Another Priority Q Data Structure: Binary Search Tree

Average performance
insert:

deleteMin:

Problems
1.

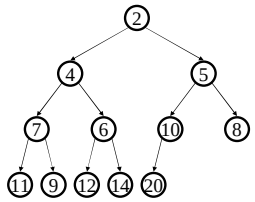
2.



13

A *Better* Priority Q Data Structure: Binary Heap

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

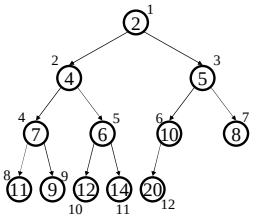


How do we find the minimum?

14

Nifty Storage Trick: Array

- Index calculations:
  - child:
  - parent:
  - root:
  - next free:

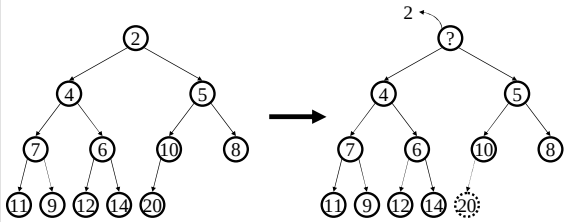


0	1	2	3	4	5	6	7	8	9	10	11	12
12	2	4	5	7	6	10	8	11	9	12	14	20

15

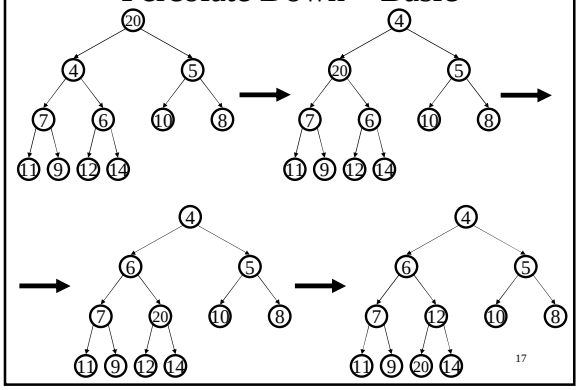
DeleteMin

`pqueue.deleteMin()`



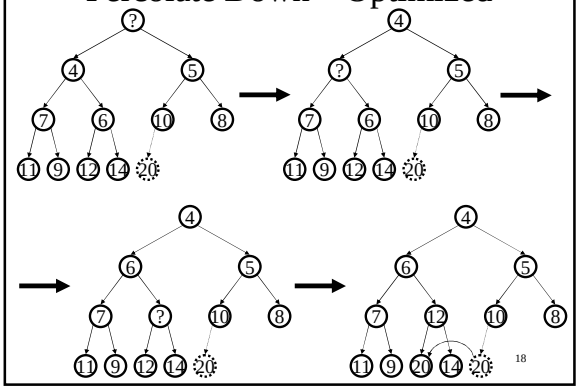
16

Percolate Down – Basic



17

Percolate Down – Optimized



18

DeleteMin Code (Optimized)

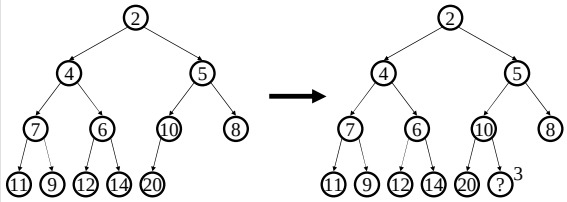
```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[1];
  size--;
  newPos =
    percolateDown(1,
         Heap[size+1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

runtime:

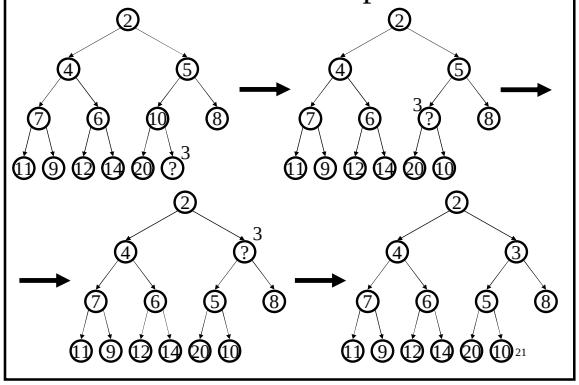
```
int percolateDown(int hole,
                  Object val) {
while (2*hole <= size) {
    left = 2*hole;
    right = left + 1;
    if (right ≤ size &&
        Heap[right] < Heap[left])
      target = right;
    else
      target = left;

    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:

Other Priority Queue Operations

- **decreaseKey**
  - given a pointer to an object in the queue, reduce its priority value

  Solution: change priority and _____
- **increaseKey**
  - given a pointer to an object in the queue, increase its priority value

  Solution: change priority and _____

Why do we need a *pointer*? Why not simply data value?

More Priority Queue Operations

- **remove**
  - given a pointer to an object in the queue, remove it

  Solution: set priority to negative infinity, percolate up to root and deleteMin
- **buildHeap**

  Naïve solution:

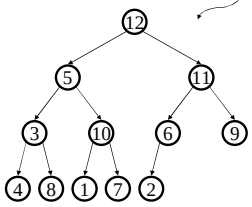
  Running time:

  Can we do better?

BuildHeap: Floyd's Method

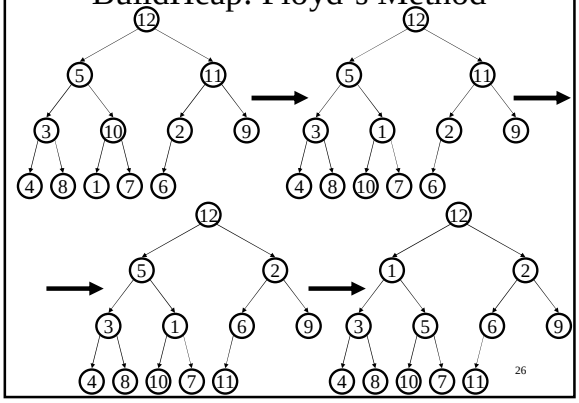
12	5	11	3	10	6	9	4	8	1	7	2

Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!



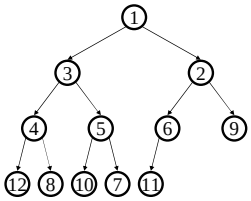
25

BuildHeap: Floyd's Method



26

Finally…



runtime:

27

Facts about Heaps

Observations

- finding a child/parent index is a multiply/divide by two
- operations jump widely through the heap
- each percolate step looks at only two new nodes
- inserts are at least as common as deleteMins

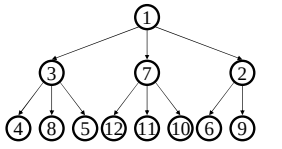
Realities

- division/multiplication by powers of two are equally fast
- looking at only two new pieces of data: bad for cache!
- with huge data sets, disk accesses dominate

28

A Solution: *d*-Heaps

- Each node has *d* children
- Still representable by array
- Good choices for *d*:
  - 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
  - optimize performance based on # of inserts/removes



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

29

Operations on *d*-Heap

- Insert : runtime =
- deleteMin: runtime =

Does this help insert or deleteMin more?
Is this good or bad?

30

One More Operation

- Merge two heaps. Ideas?

Can do in Θ(log n) worst case time.
Next lecture!

To Do

- Assignments : Project 1 – check updates!
- Reading : Chapter 6
- Admin : Sign up for class email list