CSE 332: Data Structures and Parallelism

Fall 2022

Richard Anderson

Lecture 5: Priority Queues, Part II

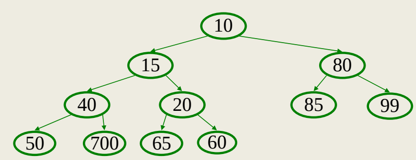
Announcements

- Reading: Weiss, for Wednesday and Friday
  - Priority Queues, 6.1-6.5
- P1 Due on Thursday, Oct 13.
- Exercise 2, due next Monday
- Longer term – beyond event horizon
  - Midterm, Friday, Nov 4
  - P2 due, Thursday, Nov 10

Priority Queues (or Heaps)

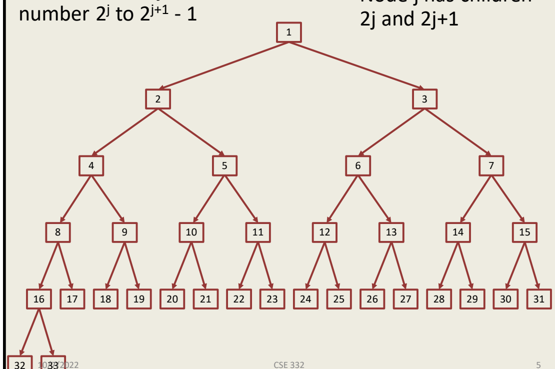
- Manage a set Insert and Delete Min
- Represent the data set as a binary tree
  - Property 1: Completeness
    - Tree is height ⌊**log n**⌋ with all leaves as far to the left as possible (for n elements in Heap)
  - Property 2: Heap Condition
    - For every non-root node X, the value of the parent of X is less than or equal to the value of X (in other words, children are bigger than their parents)

Heap operations, O(log n) time

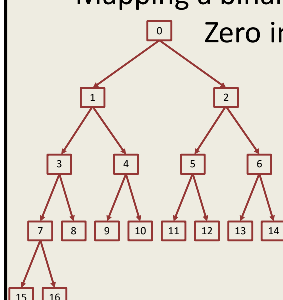


Nodes on level j are number 2^j to $2^{j+1} - 1$

Node j has children 2j and 2j+1



Mapping a binary tree to an array Zero indexing

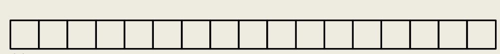


Node j has:

Left child 2j + 1

Right child 2j + 2

Parent ⌊(j-1)/2⌋



Why use an array

Insert: 16, 32, 4, 69, 105, 43, 2

0	1	2	3	4	5	6	7	8

Insert Code

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

```
int percolateUp(int hole,
                int val) {
  while (hole > 0 &&
         val < Heap[(hole-1)/2])
    Heap[hole] = Heap[(hole-1)/2];
    hole = (hole-1)/2;
  }
  return hole;
}
```


DeleteMin Code

```
int deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[0];
  size--;
  newPos =
    percolateDown(0,
        Heap[size + 1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

```
int percolateDown(int hole,
                  int val) {
while (2*hole <= size) {
    left = 2*hole + 1;
    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;
}
```


More Priority Queue Operations

decreaseKey(nodePtr, amount):

given a pointer to a node in the queue, reduce its key value

Binary heap: change priority of node and ________________

increaseKey(nodePtr, amount):

given a pointer to a node in the queue, increase its key value

Binary heap: change priority of node and ________________

Still More Priority Queue Operations

remove(objPtr):

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

Binary heap: ______________________________________

findMax( ):

Find the object with the highest value in the queue

Binary heap: ______________________________________

Building a Heap

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

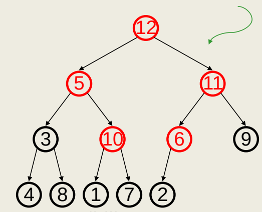
BuildHeap: Floyd's Method

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

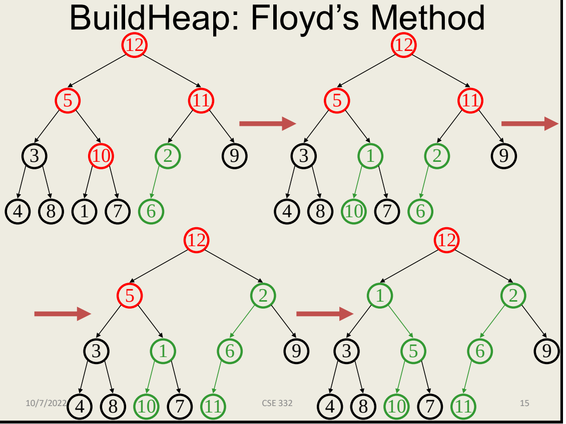
Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!

Red nodes need to percolate down

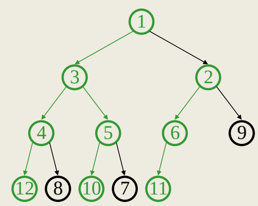
Key idea: fix red nodes from **bottom-up**



BuildHeap: Floyd's Method



Finally . . .



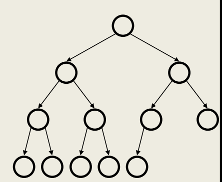
Buildheap pseudocode

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

runtime:

Buildheap Analysis

n/4 nodes percolate at most 1 level
n/8 percolate at most 2 levels
n/16 percolate at most 3 levels
...



runtime:

The Math:

$$\sum_{i \geq 1} \frac{i}{2^i} = 2$$

$$\frac{n}{4} + \frac{2n}{8} + \frac{3n}{16} + \frac{4n}{32} + \cdots = \frac{n}{2}\left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots\right] = \frac{n}{2}\sum_{i \geq 1}\frac{i}{2^i}$$

$$\begin{aligned} S = \sum_{i \geq 1}\frac{i}{2^i} &= \sum_{i \geq 1}\frac{1}{2^i} + \sum_{i \geq 1}\frac{i-1}{2^i} = 1 + \sum_{i \geq 1}\frac{i-1}{2^i} = 1 + \frac{1}{2}\sum_{i \geq 1}\frac{i-1}{2^{i-1}} \\ &= 1 + \frac{1}{2}\sum_{i \geq 0}\frac{i}{2^i} = 1 + \sum_{i \geq 1}\frac{i}{2^i} = 1 + \frac{S}{2} \end{aligned}$$

Heap Sort

```
HeapSort(int[] A){
  BuildHeap(A);
  for (int i = A.Length - 1; i >= 0; i--){
    A[i] = DeleteMin(A);
  }
}
```

This version sorts in decreasing order – for increasing order, either reverse the result, or use a MaxHeap.

Why Heapsort is great

- Relatively easy to code
- O(n log n) worst case runtime
- In place
- Elegant use of space to store results as heap shrinks