CSE 332: Data Structures and Parallelism

Fall 2022

Richard Anderson

Lecture 4: Priority Queues

Announcements

- Reading: Weiss, for Wednesday and Friday
  - Priority Queues, 6.1-6.5
- Checkpoint for P1 on Thursday.
- Exercise 2, due next Monday
- Quiz section
  - Big OH, Algorithm run time analysis, Heaps

Today

- Priority Queues
- Binary Tree Implementation
- Array implementation

Summary of Monday

- We need a rigorous way of talking about the performance of an algorithm
  - Real world to math world
- Ideas
  - Measure run time on an input by counting "steps"
  - Run time function for an algorithm, R(n): Worst case run time on input of size n
  - Only consider the rate of growth of run time functions by ignoring constants with big-Oh

Change of base of logs

$$\log_a N = \log_a(b^{\log_b N}) = \log_b N \log_a b$$

$$\log_b N = \frac{\log_a N}{\log_a b}$$

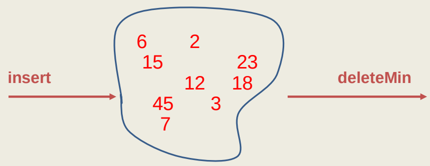
Base of the log can be ignored in big-Oh (provided the base is a constant)

Priority Queues

- Manage a set, with insert item and get highest priority item
  - We will work with "MinHeaps" – where the smallest key has the highest priority
    - Straight forward changes to handle MaxHeaps
  - This is interesting in the case where inserts and selects are intermixed

Priority Queue ADT

- Need a new ADT
- Operations: Insert an Item,
  Remove the "Best" Item



Priority Queue ADT

1. PQueue data : collection of data with priority

2. PQueue operations
   – insert
   – deleteMin

   (also: create, destroy, is_empty)

3. PQueue property: if *x* has a smaller key than *y*, *x* will be returned before *y*

Potential Implementations

	Insert	DeleteMin
Unsorted list (Array)		
Unsorted list (Linked list)		
Sorted list (Array)		
Sorted list (Linked list)		
Binary Search Tree		

Binary Heap data structure

- Binary Heap (a kind of binary tree) for priority queues:
  – O(log n) worst case for both insert and deleteMin

- It's optimized for priority queues. Lousy for some other types of operations (e.g., searching)

Tree Review

root(T): A

leaves(T): D-F, I-N

children(B): D-F

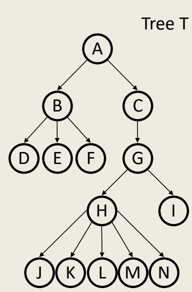
parent(H): G

siblings(E): D,F

ancestors(F):

descendants(G):

subtree(C):



More Tree Terminology

depth(B):

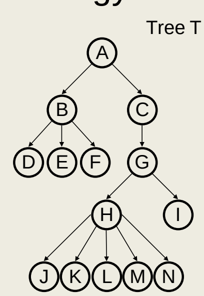
height(G):

height(T):

degree(B):

branching factor(T):

n-ary tree:



Binary Heap Properties

A binary heap is a binary tree with two important properties that make it a good choice for priority queues:

1. Completeness
2. Heap Order

Completeness

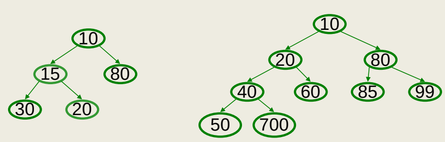
A binary heap is a complete binary tree

All levels are full, except the bottom, which is filled to the right

Completeness guarantees that heap with n items is a tree with height log n

Heap Order Property

For every non-root node X, the value of the parent of X is less than or equal to the value of X



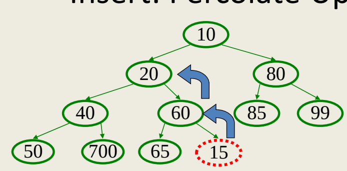
Heap Operations

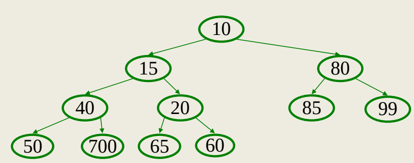
- Main operations: insert, deleteMin
- Key is to maintain
  - Completeness
  - Heap Order
- Basic idea is to propagate changes up/down the tree, fixing order as we go

Insert (val)

- Create a new leaf at the bottom of the tree for val
- Percolate up by exchange with parent as long as is needed

Insert: Percolate Up



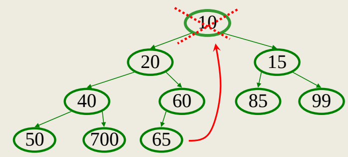


DeleteMin

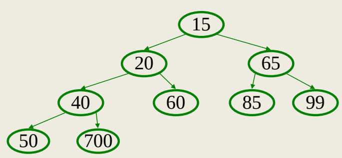
Basic Idea:

1. Remove min element (the root)
2. Put “last” leaf node value at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

DeleteMin: Percolate Down

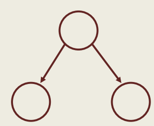


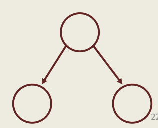
DeleteMin: percolate down



Correctness Proofs

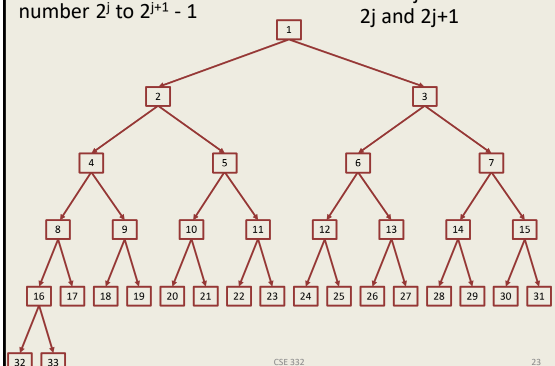
- Show operations preserve heap properties
- Insert
  - Complete tree
  - Parent smaller than children
- DeleteMin
  - Return smallest element
  - Complete tree
  - Parent smaller than children



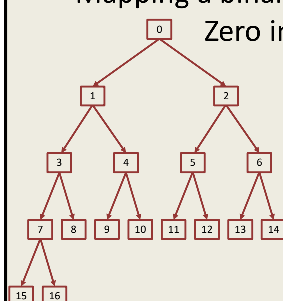


Nodes on level j are number 2^j to $2^{j+1} - 1$

Node j has children 2j and 2j+1



Mapping a binary tree to an array Zero indexing

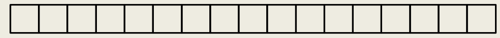


Node j has:

Left child 2j + 1

Right child 2j + 2

Parent $\lfloor (j-1)/2 \rfloor$



Why use an array

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;
}
```


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

0	1	2	3	4	5	6	7	8

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

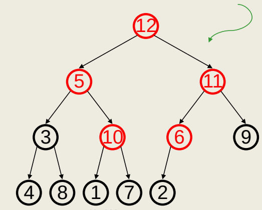
BuildHeap: Floyd's Method

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

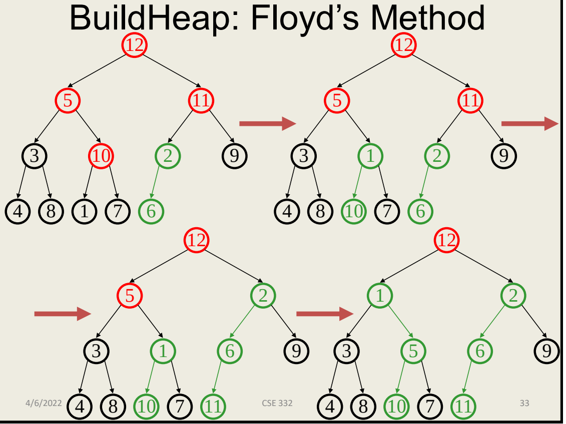
Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!

Red nodes need to percolate down

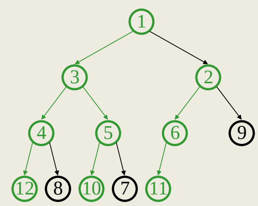
Key idea: fix red nodes from **bottom-up**



BuildHeap: Floyd's Method



Finally . . .



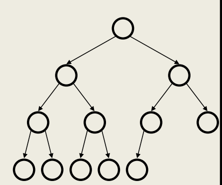
Buildheap pseudocode

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

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

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