**BINARY HEAPS**

**Two Additional Properties**

- **“Structure” property:**
  - Levels are “filled” in order, left to right
  - Also known as complete binary trees
    - All levels (except possibly bottom) completely filled
    - Bottom level filled from left to right

- **“Heap order” property:**
  - Key in parent of X \( \leq \) key in X
    - Only defined for Comparable objects
  - Where would the minimum be?
  - Where would the maximum be?
Are These Binary Heaps?

Binary Heaps: Structural Properties

- How many nodes does a binary heap with height \( h \) have?
  - \( \geq 2^h \)
  - \( \leq 2^{h+1} - 1 \)

- What is the height of a binary heap with \( N \) nodes?
  - \( \text{floor}(\log_2 N) \)
Binary Heaps: Array Implementation

- Structure property ➔ heaps are complete binary trees
  - No “holes” in any level (including bottom)
  - Levels are filled in order (left-to-right)
- Good match for array implementation
  - Without wasting space
- Root at A[0]
- A[x] has left child at A[2x+1]
- A[x] has right child at A[2x+2]
- A[x] has parent at A[(x-1)/2]

```
A   0  1  2  3  4  5  6  7  8  9 10 11 12 13
  13 21 16 24 31 19 68 65 26 32
```

**Binary Heaps: Array Implementation**

- Structure:
  - Root at A[0]
  - A[x] has left child at A[2x+1]
  - A[x] has right child at A[2x+2]
  - A[x] has parent at A[(x-1)/2]

- Optimization – move everything right by 1
  - Root at A[1]
  - A[x] has left child at A[2x]
  - A[x] has right child at A[2x+1]
  - A[x] has parent at A[x/2]

```
A   0  1  2  3  4  5  6  7  8  9 10 11 12 13
  13 21 16 24 31 19 68 65 26 32
```
Binary Heap: Class Structure

```java
public class BinaryHeap<AnyType extends Comparable<? Super AnyType>> {
    private AnyType[] array;
    private int currentSize;

    public BinaryHeap(int capacity) {
        currentSize = 0;
        array = (AnyType[]) new Comparable[capacity+1];
    }

    public AnyType min() { ... }
    public void insert(AnyType x) { ... }
    public AnyType deleteMin() { ... }
    public AnyType isEmpty() { return currentSize == 0; }
    public AnyType isFull(){return currentSize == array.length-1;}
}
```

Binary Heaps: Basic Operations

- Required operations: min, insert, deleteMin
- min: easy!
  - Just read the element stored at the root!
    - // PRE: heap is not empty
      - public AnyType min() { return array[1]; }
  - Run-time?
- Also easy to perform the remaining two operations:
  - Mainly ensure that the two heap properties are maintained
Binary Heaps: \textit{insert}

- To insert element X to heap:
  - Create a hole at next available location
    - To maintain “structure” property
  - If “heap-order” allows, placed X in hole
  - Else, bubble hole up toward the root
    - Until X can be placed in hole

- Example: insert 14

Implementing \textit{insert}

// PRE: heap is not full
public void insert(AnyType x) {
    // Create hole
    currentSize++;
    int hole = currentSize;

    // Percolate up
    array[0] = x;
    for( ; x.compareTo(array[hole/2]) < 0; hole = hole/2)
        array[hole] = array[hole/2];
    array[hole] = x;
}

- Complexity of \textit{insert}?
  - \(O(\log N)\) – arrays help with \(O(1)\) moves between levels
Binary Heaps: deleteMin

- Finding the min is easy – how to remove it?
  - When min is removed, hole is created at root
  - Since heap size reduce by 1, last element X must be moved somewhere
    - Unlikely that X can be moved to hole at root, though
  - “Percolate down” the hole –
    - By sliding smaller of the hole’s children into it
    - Repeat until X can be placed in the hole

→ effectively, place X in correct spot, along a path from root containing minimum children
Implementing `deleteMin`

// PRE: heap is not empty
public AnyType deleteMin() {
    // min value to be returned
    AnyType minItem = array[1];
    // Move last item to root
    array[1] = array[currentSize];
    currentSize--;
    // And percolate it down to the right
    percolateDown();
    return minItem;
}

Implementing `percolateDown`

private void percolateDown() {
    tmp = array[1];
    int hole = 1;
    int child; // smaller child of array[hole]
    for( ; hole*2 <= currentSize; hole = child) {
        child = hole*2;
        if (child != currentSize &&
            array[child+1].compareTo(array[child]) < 0 )
            child++;
        if (array[child].compareTo(tmp) < 0)
            array[hole] = array[child];
        else break;
    }
    array[hole] = tmp;
}

Complexity of `deleteMin`?
Binary Heaps: Other Operations

- Basic operations
  - min: $O(1)$
  - insert: $O(\log N)$
  - deleteMin: $O(\log N)$

- Additional operations:
  - decreaseKey($p$, $\Delta$) – lower the value at position $p$ by $\Delta$
    - lower value, then “percolate up” to maintain heap order
  - increaseKey($p$, $\Delta$) – increase value at position $p$ by $\Delta$
    - “Percolate down” to maintain heap order
  - delete($p$) – remove node at position $p$
    - decreaseKey($p$, infinity); deleteMin();
  - buildHeap(): build heap from an initial collection of items
    - How?

Binary Heaps: buildHeap

- Use N successive calls to insert:
  - Worst-case – $O(N \log N)$

- Can we do better?
  - Place N items in an unsorted array
    - While maintaining the structure property
  - For all nodes $i = N/2$, ..., 1, percolate them down one-by-one

150, 80, 40, 30, 10, 70, 110, 100, 20, 90, 60, 50, 120, 140, 130
Binary Heaps: buildHeap

- For all nodes \( i = \frac{N}{2}, \ldots, 1 \), percolate them down one-by-one