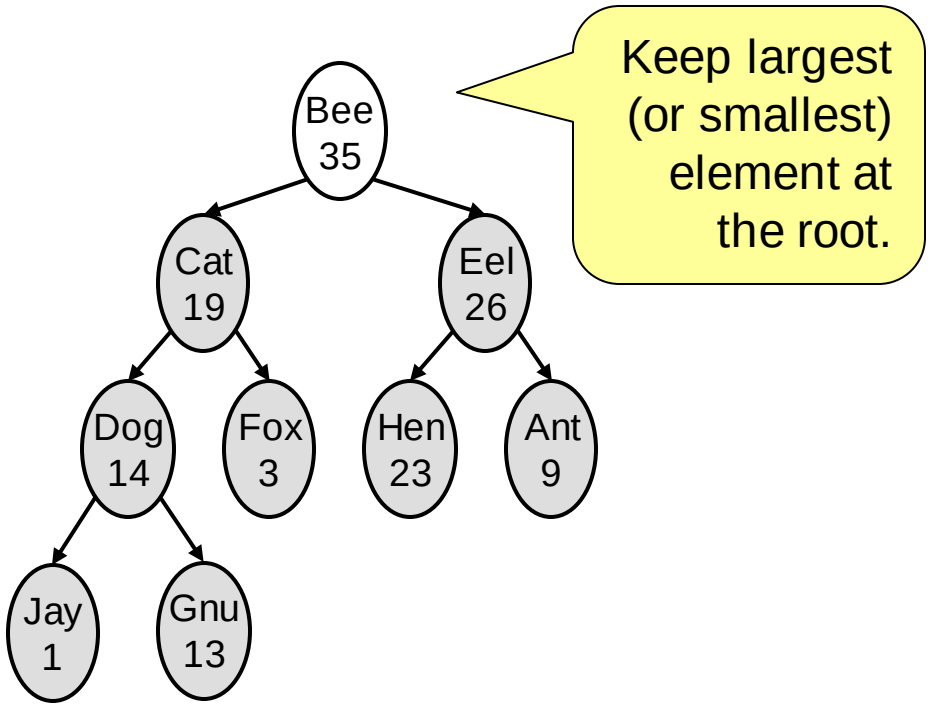
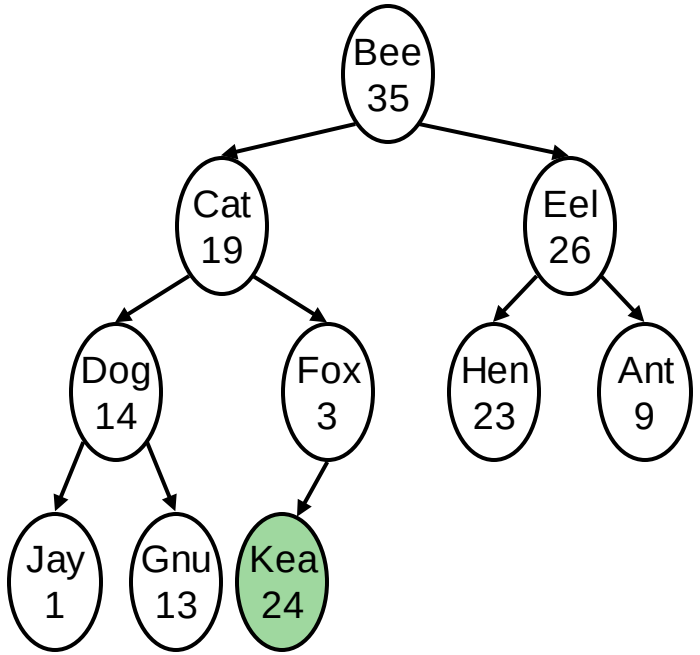
Partially Ordered Trees

- **Partially Ordered Tree - implementing Priority Queues efficiently**
- Binary tree
- Children ≤ parent,
- Order of children not important



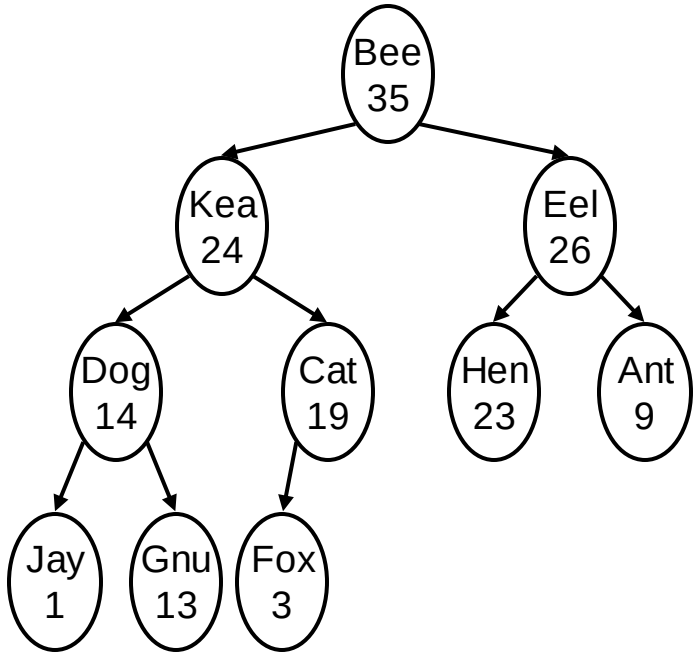
Partially Ordered Tree: add

- Easy to add and remove because the order is not complete.
- Add:
  - insert at bottom rightmost
  - “push up” to correct position.
    (swapping)



Partially Ordered Tree: remove

- Easy to add and remove because the order is not complete.
- Add:
  - insert at bottom rightmost
  - "push up" to correct position.
- Remove:
  - "pull up" largest child and recurse.
  - But: makes tree unbalanced!

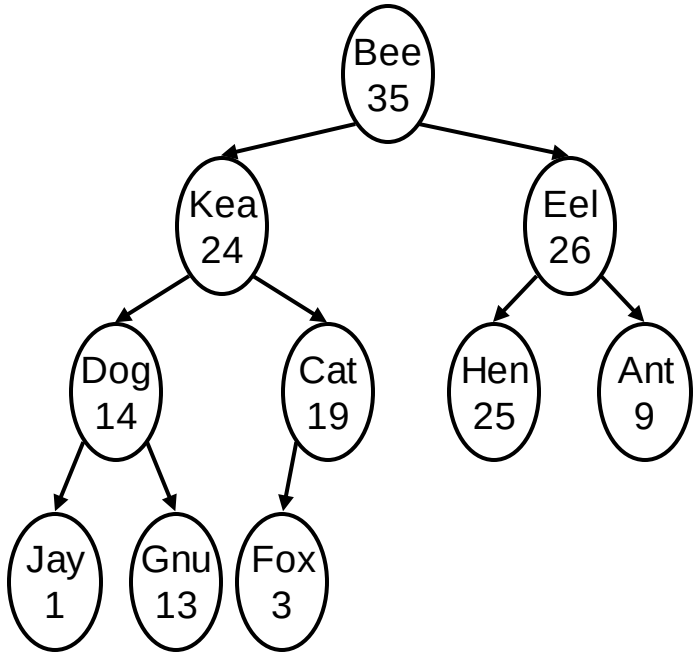


Partially Ordered Tree: remove I

- Easier to add and remove because the order is not complete.
- Add:
  - insert at bottom rightmost
  - "push up" to correct position.
- Remove:
  - "pull up" largest child of root and recurse on that subtree.
  - But: makes tree unbalanced!

Alternative:

- replace root by bottom rightmost node
- "push down" to correct position (swapping)
- keeps tree balanced – and complete!

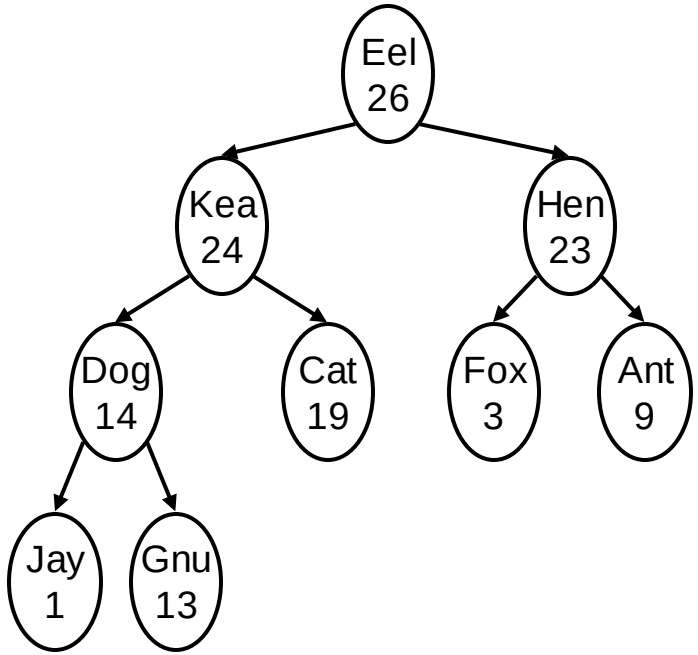


Partially Ordered Tree: remove II

- Easier to add and remove because the order is not complete.
- Add:
  - insert at bottom right
  - “push up” to correct position.
- Remove:
  - “pull up” largest child and recurse.
  - But: makes tree unbalanced!

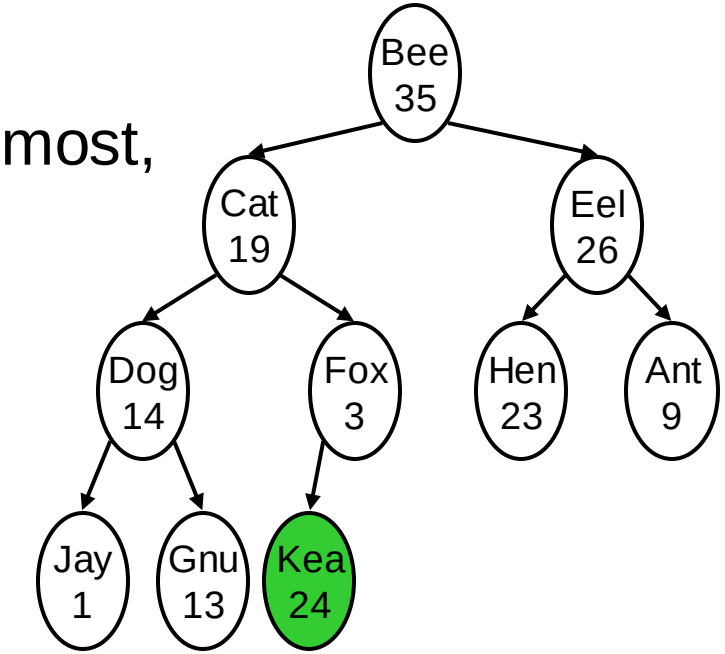
Alternative:

- replace root by bottom rightmost node
- “push down” to correct position
- keeps tree balanced – and complete!



Partially Ordered Tree

- Add: insert at bottom rightmost, swap with parent, …
- Remove: replace root with bottom rightmost, swap with largest child, …



But:

- How do you find the bottom right?
- Once you have found it, how do you find its parent to push it up?

We need a tree where you can quickly get to:

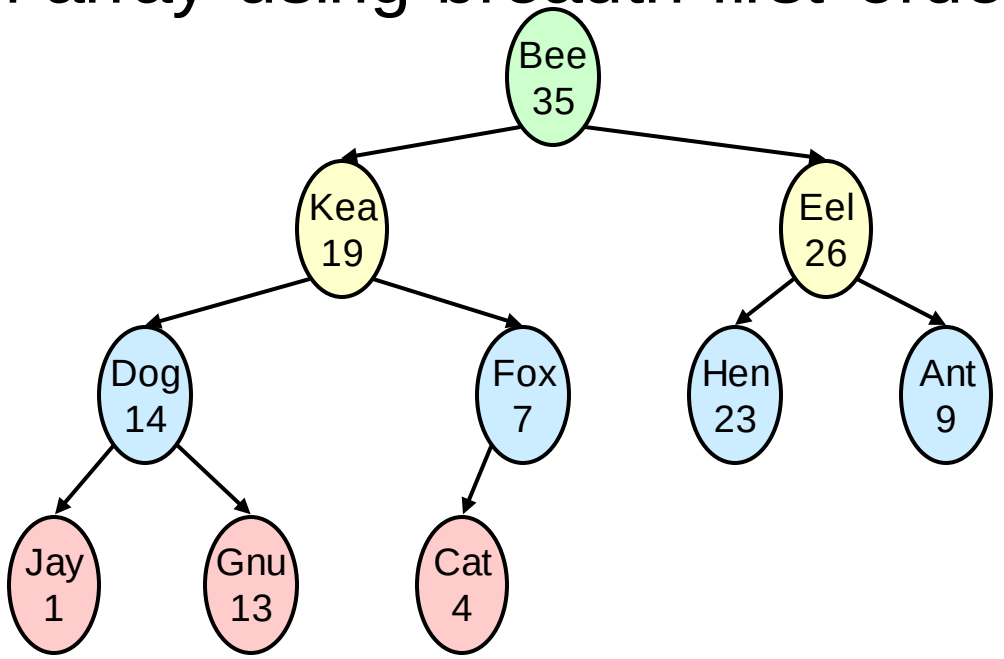
- the bottom right node,
- children from parent,
- parent from children.

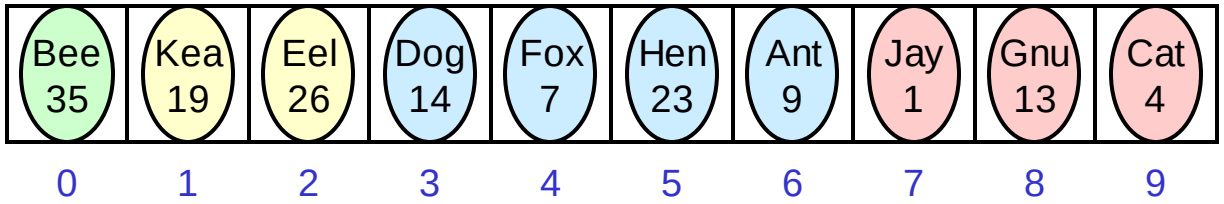
Heap:

- A complete, partially ordered, binary tree

  complete = every level full, except bottom, where nodes are to the left

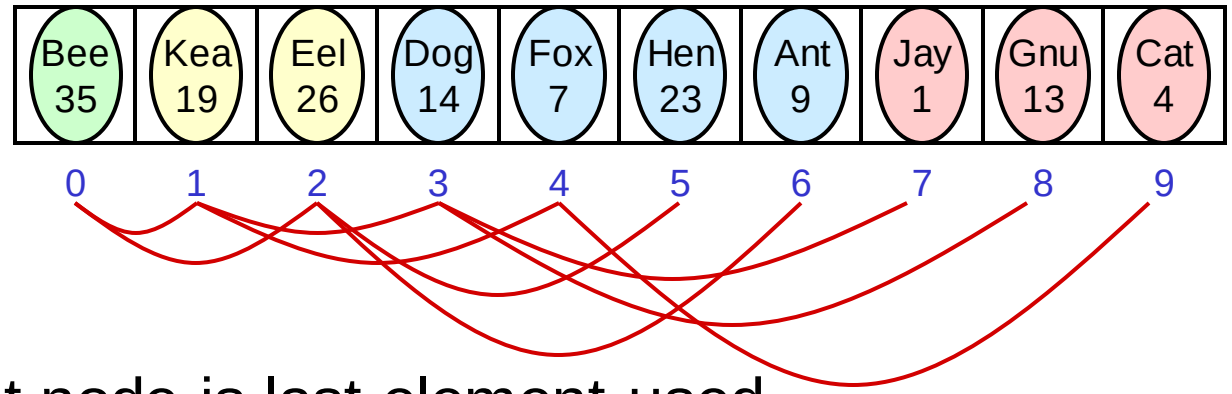
- Implemented in an array using breadth-first order





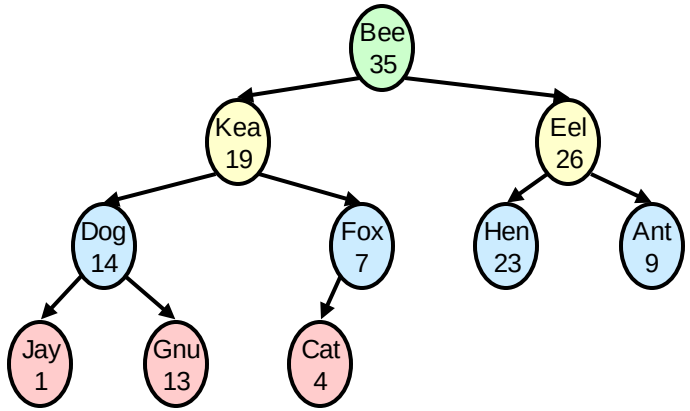
Heap

- We can **compute the index** of parent and children of a node:
  - the children of node i are at (2i+1) and (2i+2)
  - the parent of node i is at (i-1)/2

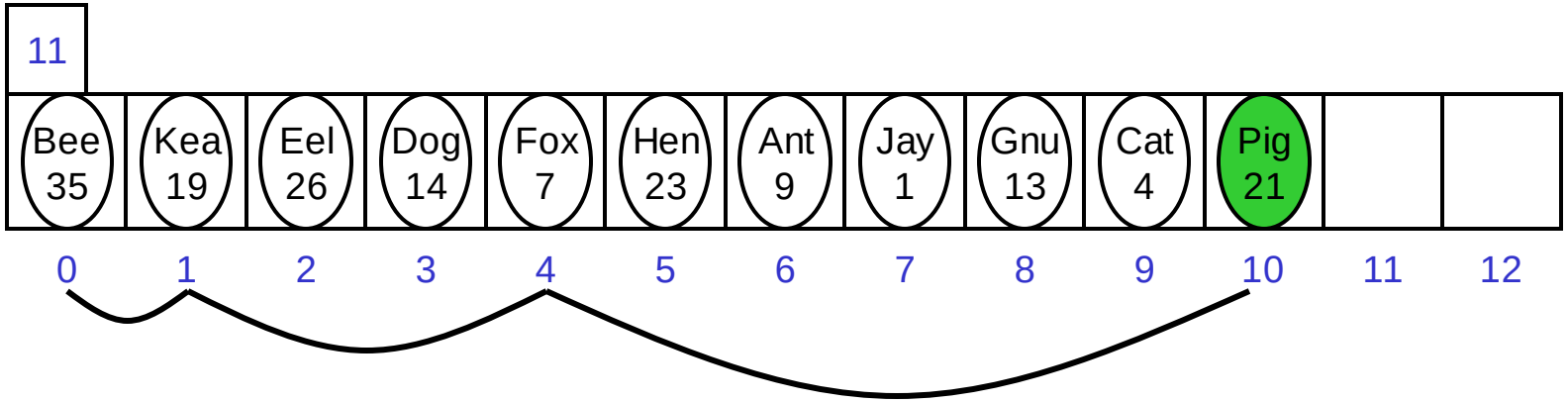


- Bottom right node is last element used.

- There are no gaps!



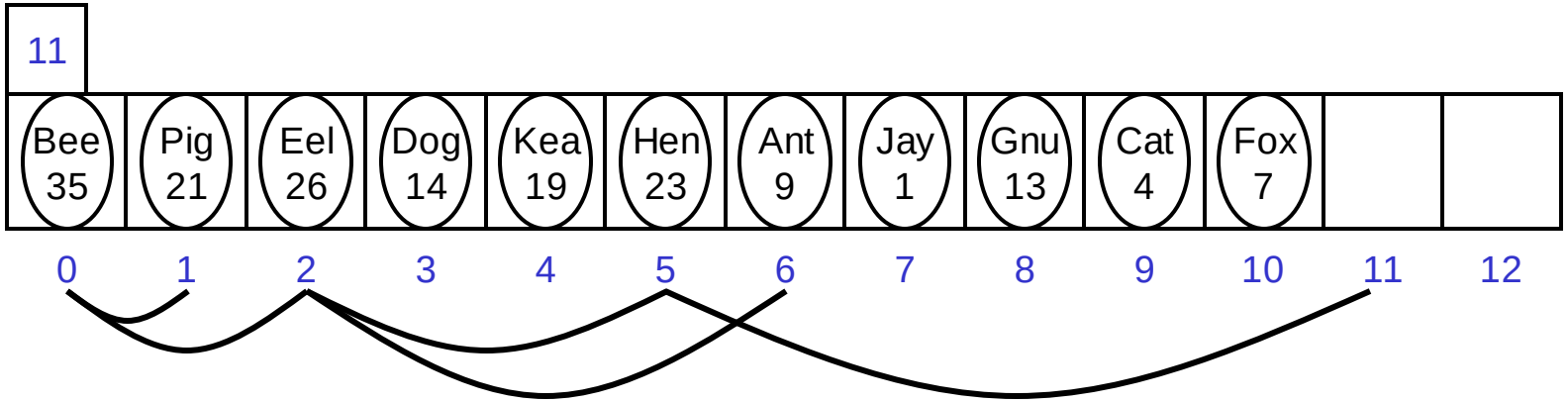
Heap: add



Insert at bottom of tree and push up:

- Put new item at end: 10
- Compare with parent: (10-1)/2 = 4 ⇒ Fox/7
  - If larger than parent, swap
- Compare with parent: (4-1)/2 = 1 ⇒ Kea/19
  - If larger than parent, swap
- Compare with parent: (1-1)/2 = 0 ⇒ Bee/35

Heap: remove



- Remove item at 0:
- Move last item to 0
- Find largest child $2\times0+1 = 1$, $2\times0+2 = \mathbf{2}$
  - If smaller than largest child, swap
- Find largest child $2\times2+1 = 5$, $2\times2+2 = 6$
  - If smaller than largest child, swap
- Find largest child $2\times5+1 = 11$: No such child

HeapQueue

```
public class HeapQueue <E> extends AbstractQueue <E> {
    private List<E> data =  new ArrayList<E>();
    private Comparator<E> comp;
    public HeapQueue (Comparator <E> c) {
        comp = c;
    }

    public boolean isEmpty() {
        return data.isEmpty();
    }
    public int size () {
        return data.size();
    }
    public E peek () {
        if  (isEmpty())     return null;
        else                return data.get(0);
    }
```

Use ArrayList, not array, so it handles resizing.

Comparator must be designed so that it compares the priority values (not the items)

HeapQueue: offer and poll

```
public boolean offer(E value) {
    if  (value == null)       return false;
    else  {
        data.add(value);
        pushup(data.size()-1);
        return true;
    }
}
```

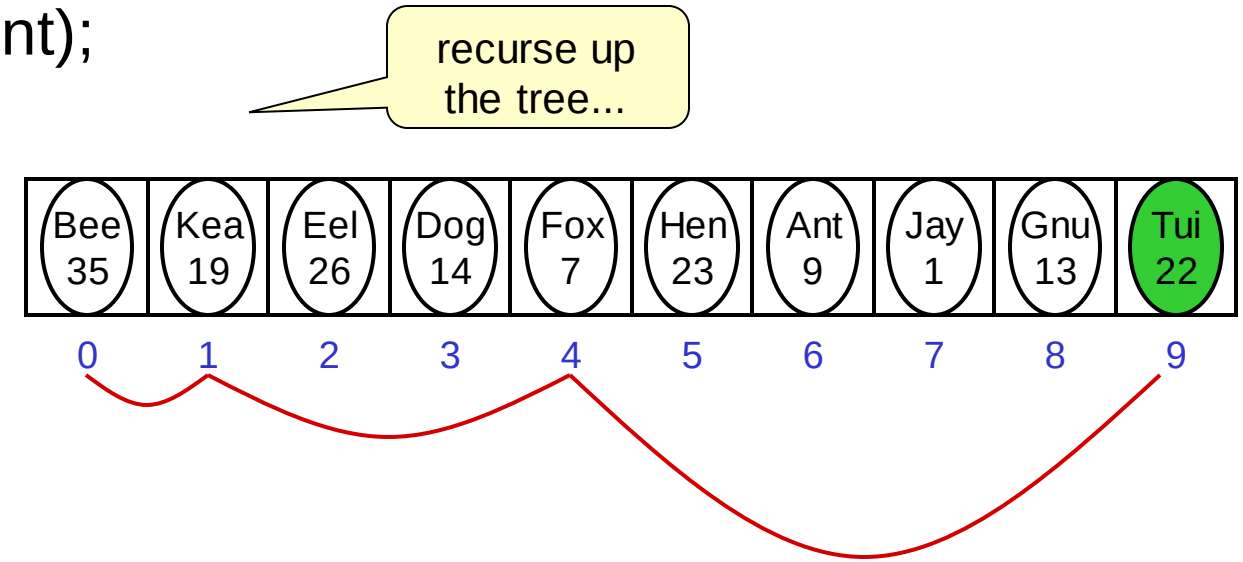
add at the end of the array

```
public E poll() {
    if  (isEmpty())           return null;
    if  (data.size() == 1)    return data.remove(0);
    else  {
        E ans = data.get(0);
        data.set(0, data.remove(data.size()-1));
        pushdown(0);
        return ans;
    }
}
```

move last element into root

HeapQueue: pushup

```
private void pushup(int child) {
    if (child == 0) return;
    int parent = (child-1)/2;
    // compare with value at parent and swap if parent smaller
    if (comp.compare(data.get(parent), data.get(child)) < 0) {
        swap(data, child, parent);
        pushup(parent);
    }
}
```

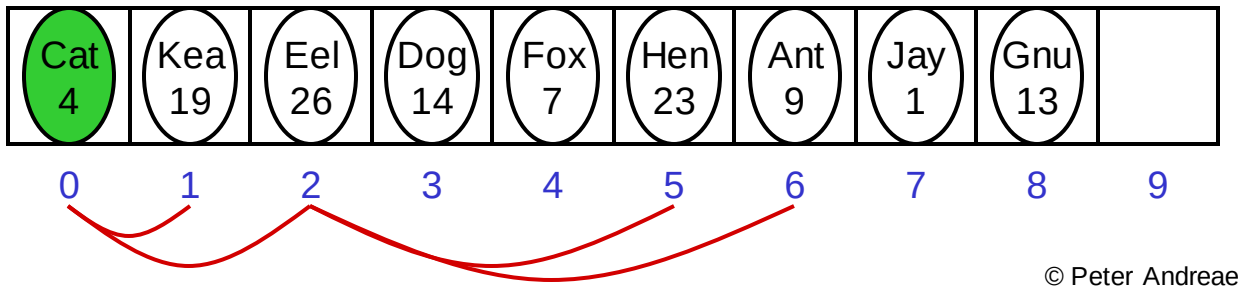



```
private void swap(List<E> data, int from, int to)
  data.set(child, data.set(parent, data.get(child)));
```

HeapQueue: pushdown

```
private void pushdown(int parent) {
    int largeCh = 2*parent+1;
    int otherCh = largeCh+1;
    // check if any children
    if (largeCh >= data.size()) return;
    // find largest child
    if (otherCh < data.size() &&
        comp.compare(data.get(largeCh), data.get(otherCh)) < 0 )
        largeCh = otherCh;
    // compare with largest child, and swap if smaller
    if (comp.compare(data.get(parent), data.get(largeCh)) < 0) {
        swap(data, largeCh, parent);
        pushdown(largeCh);
    }
}
```

recurse down the tree...



HeapQueue: Analysis

- Cost of offer:
  = cost of pushup
  = O(log(n))
    log(n) comparisons, 2 log(n) assignments

- Cost of poll:
  = cost of pushdown
  = O(log(n))
    2 log(n) comparisons, 2 log(n) assignments

- Conclusion: HeapQueue is always fast!!

- SlideShow, Sokoban
  - ArrayLists, Stacks
- WellingtonTrains, MoleculeRenderer
  - Maps, sorting, comparing
- Hospital
  - Queues/Priority Queues, simulation
- Measuring Queues, DecisionTrees
  - complexity, binary trees
- OrganisationChart, GeneSearch
  - general trees, algorithms
- CPNCalculator, MazeFinder, Permutations
  - Trees, graphs, recursion, algorithms