Data Structures and Algorithms

XMUT-COMP 103 - 2024 T1

Graphs and Heaps

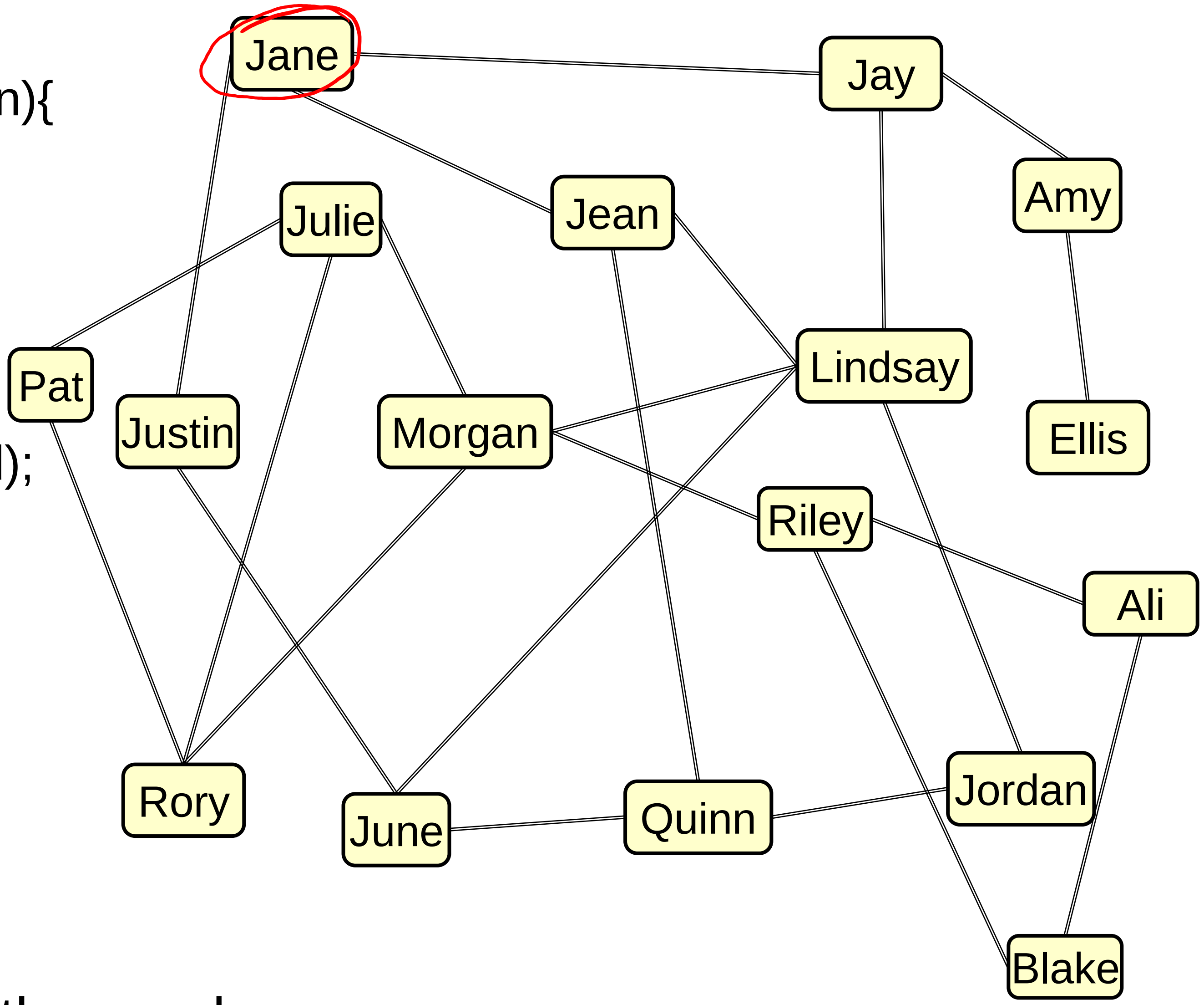
A/Prof. Pawel Dmochowski

School of Engineering and Computer Science

Victoria University of Wellington

Traversing Graphs : count connected nodes

```
/** Find number of friends in network */

public int countConnected(SNPerson person){
    person.visit();
    int count =1;
    for (Person friend : person){
        if ( ! friend.isVisited()) {
            count += countConnected(friend);
        }
    }
    return count;
}
```



Traversing a graph makes a tree within the graph.

Traversing Graphs: connectedTo

```
/** Are two people connected in the network */

public boolean connectedTo(SNPerson person, SNPerson query){
    if (person.equals(query) ) {
        return true;
    }
    person.visit();
    for (Person friend : person){
        if ( ! friend.isVisited()  &&  connectedTo(friend, query) ) {
            return true;
        }
    }
    return false;
}
```

Note: need to reset all the visited flags before you call this!

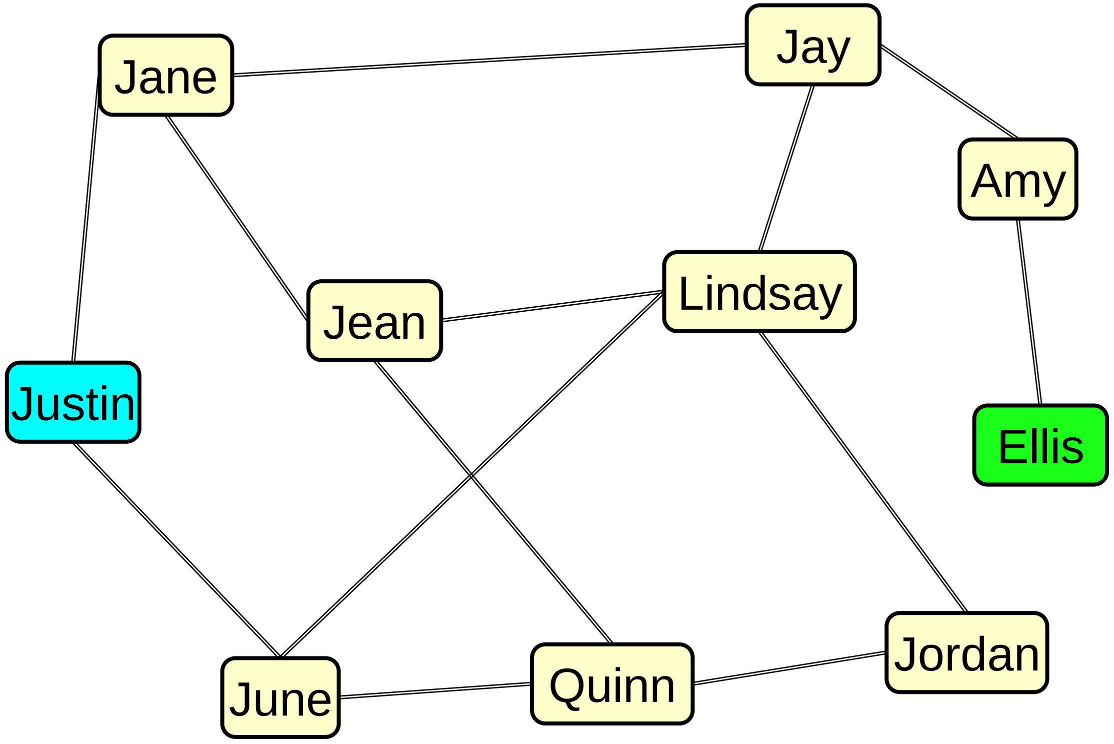
Traversing Graphs: connectedTo

```
/** Are two people connected in the network */

public boolean connectedTo(SNPerson person, SNPerson query){
    return connectedTo(person, query, new HashSet<SNPerson>());
}

public boolean connectedTo(SNPerson person, SNPerson query, Set<SNPerson> visited){
    if (person.equals(query) ) {
        return true;
    }
    visited.add(person);
    for (Person friend : person){
        if ( ! visited.contains(friend)  &&  connectedTo(friend, query, visited) ) {
            return true;
        }
    }
    return false;
}
```

Traversing Graphs: connectedTo

```
/** Are two people connected in the network */
public boolean connectedTo(SNPerson person, SNPerson query){
    return connectedTo(person, query, new HashSet<SNPerson>());
}
public boolean connectedTo(SNPerson person, SNPerson query, Set<SNPerson> visited){
    UI.println(person);
    if (person.equals(query) ) {  return true; }
    visited.add(person);
    boolean ans = false;
    for (Person friend : person){
        if ( ! visited.contains(friend)  &&
                connectedTo(friend, query, visited) ) {
            ans = true;
        }
    }
    visited.remove(person);
    return ans;
}
```

What happens if we unvisited the node here?



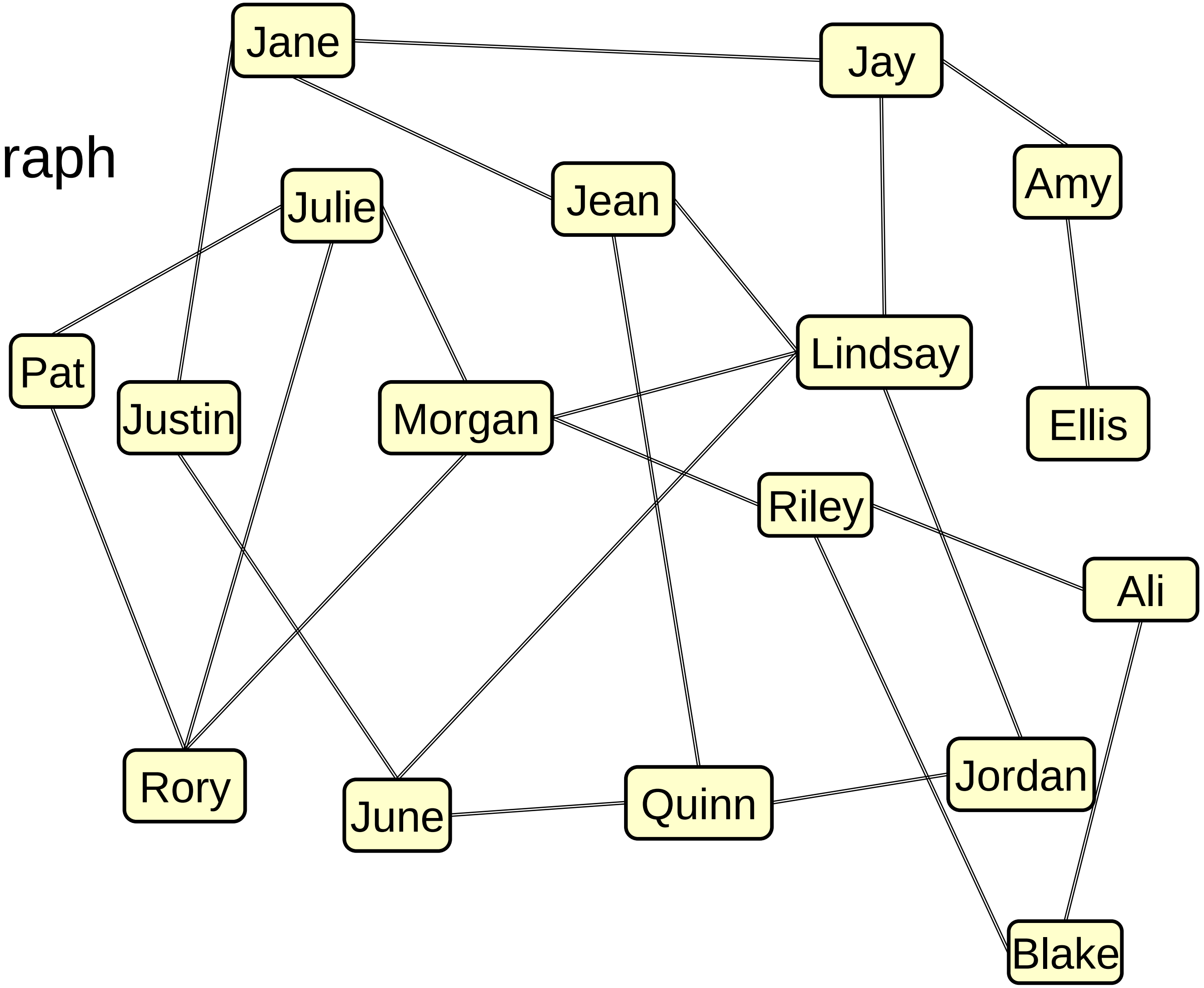
Traversing Graphs: iterative

```
/** Are two people connected in the network */

public boolean connectedTo(SNPerson person, SNPerson query){
    Stack<SNPerson> stack = new ArrayDeque<SNPerson>();
    Set<SNPerson> visited = new HashSet<SNPerson>();
    stack.push(person);
    while (! stack.isEmpty()){
        SNPerson p = stack.pop();
        visited.add(p);
        if (p.equals(query) ) { return true; }
        for (Person friend : p){
            if ( ! visited.contains(friend)) { stack.push(friend); }
        }
    }
    return false;
}
```

Is Graph Connected?

- How do I represent this graph?
- Just having one node isn’t enough!
- Need to store the set of nodes in the graph

```
private Set<SNPerson> graph;
```



Is it connected?

pick a node,

find all the connected nodes

does this cover all the nodes?

Searching for Items in a sorted List.

- Searching for an item in a List is normally O(n)   (contains, indexOf)
- If the List is sorted, we can do much better.

- Binary Search: Finding "Gnu"

10

0	1	2	3	4	5	6	7	8	9
Ant	Bee	Cat	Dog	Fox	Gnu	Owl	Pig	Rat	Tui

- Look in the middle:
  - if item is middle item ⇒ return
  - if item is before middle item ⇒ look in left half
  - if item is after middle item ⇒ look in right half

Binary Search (recursive)

```
public int indexOf(String value, List<String> data){
    return indexOf(value, data, 0, data.size());
}

public int indexOf(String value, List<String> data, int low, int high){
    // value in [low .. high)  (if present)
    if (low >= high){  return -1;  }                                    //  value not present
    int mid  =  (low + high) / 2;
    int comp =  value.compareTo(data.get(mid));
    if (comp == 0)        { return mid;   }                             // item is present
    else if (comp < 0)   { return indexOf(value, data, low, mid); }     // item in [low .. mid)
    else                  { return indexOf(value, data, mid+1, high);}  // item in [mid+1 .. high)
}
```

Binary Search (recursive)

Cost:

- each recursive call cuts the range in half.
- number of recursive calls = number of times can cut n items in half = $\log_2(n)$
- cost of each line (except recursive calls) = O(1)
- Total cost = O(log(n))

```
public int indexOf(String value, List<String> data, int low, int high){
    // value in [low .. high)  (if present)
    if (low >= high){  return -1;  }                            //  value not present
    int mid  =  (low + high) / 2;
    int comp =  value.compareTo(data.get(mid));
    if (comp == 0)       { return mid;   }                      // item is present
    else if (comp < 0)   { return indexOf(value, data, low, mid); }    // item in [low .. mid)
    else                 { return indexOf(value, data, mid+1, high);}  // item in [mid+1 .. high)
}
```

Binary Search (iterative)

```
private int indexOf(String value, List<String> data){
    int low = 0;
    int high = data.size();
    // item in [low .. high)  (if present)
    while (low < high){
        int mid  =  (low + high) / 2;
        int comp =  value.compareTo(data.get(mid));
        if (comp == 0)                              // item is at mid
            return mid;
        if (comp < 0)                               // item in [low .. mid)
            high = mid;                             // item in [low .. high)
        else                                        // item in [mid+1 .. high)
            low = mid + 1;                          // item in [low .. high)
    }
    return -1;                                      // item in [low .. high) and low >= high,
}                                                   // therefore item not present
```

Another form of Binary Search

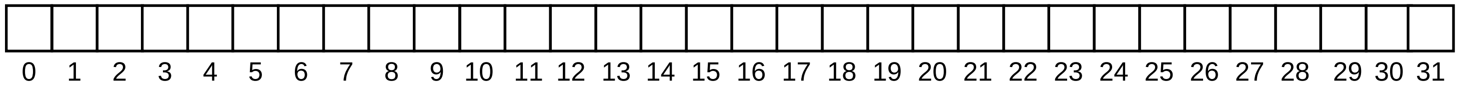
/* Return the index of where the item *ought* to be, whether present or not. (!) */

```
private int findIndex(String value, List<String> data){
  int low = 0;
  int high  =  data.size();                          // index in [low .. high]
  while (low < high){
     int mid  =  (low + high) / 2;
     if (value.compareTo(data.get(mid)) > 0)         // index in [mid+1 .. high]
         low = mid + 1;                              // index in [low .. high]  low <= high
     else                                            // index in [low .. mid]
         high = mid;                                 // index in [low .. high], low<=high
  }
  return low;                                        // index in [low .. high] and low = high
                                                     // therefore index = low
}
```

Note: correct position might be at end (index =size)

Binary Search: Cost

- What is the cost of searching if *n* items in set?
  - key step = ?



- Iteration / Size of range / Cost of iteration

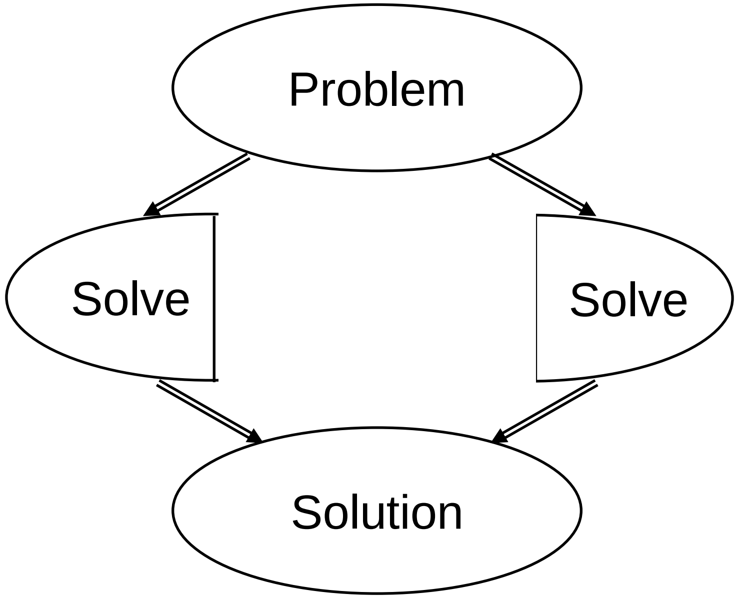
Iteration	Size of range	Cost of iteration
1	n	
2		
k	1	

$\log_2(n)$ or $\lg(n)$:

The number of times you can divide a set of *n* things in half.

lg(1000) ≈10, lg(1,000,000) ≈ 20, lg(1,000,000,000) ≈ 30

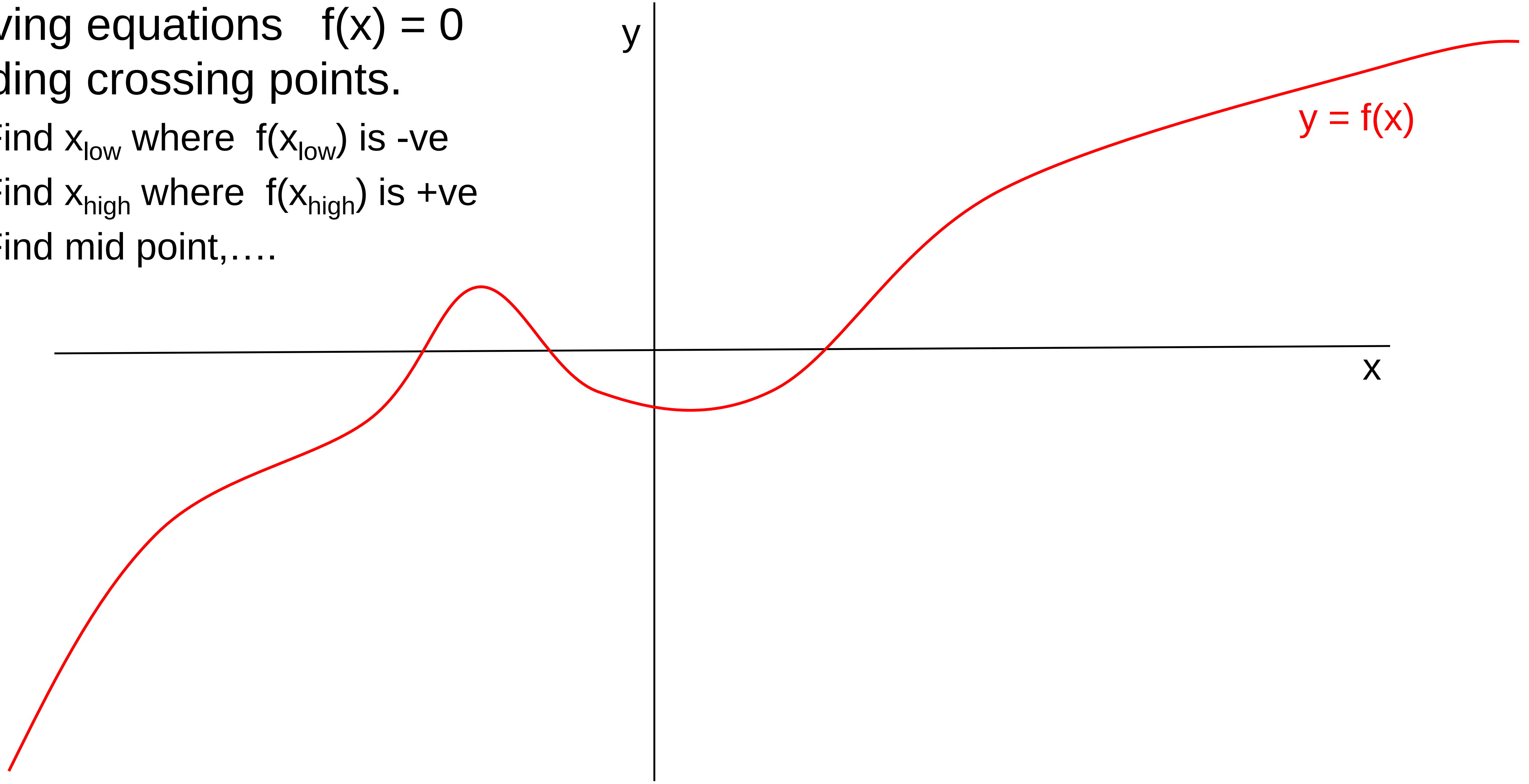
Every time you double *n*, you add one *lg(n)*



- Arises all over the place in analysing algorithms
- "Divide and Conquer" algorithms
  - Good sorting algorithms
  - binary search (sort of)
- Height of binary trees:
  - Binary tree of height h has at most 2^h -1 nodes (h = number of levels)
  - Binary tree with n nodes has height at least $\log_2(n)$

Bisection Algorithm

- Solving equations   f(x) = 0
  Finding crossing points.
  - Find x_{low} where $f(x_{low})$ is -ve
  - Find x_{high} where $f(x_{high})$ is +ve
  - Find mid point,….



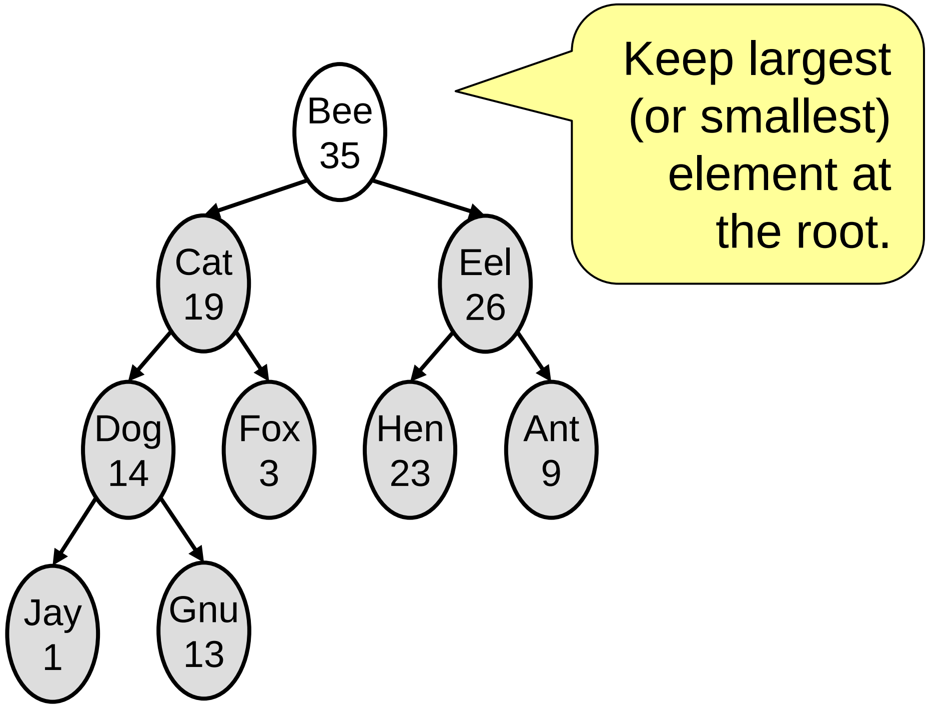
Bisection

```
bisection( -100, 100, (double x)-> {return (3*x*x*x - 4*x*x + 321);} );
bisection( -100, 100, (x)-> (3*x*x*x - 4*x*x + 321) );

public double bisection(double low, Double high, Function<Double, Double> function){
    double fLow = function.apply(low);
    double fHigh = function.apply(high);
    if (Math.abs(fLow)<THETA)  { return fLow; }
    if (Math.abs(fHigh)<THETA) { return fHigh; }
    if (Math.signum(fLow) == Math.signum(fHigh)) { return Double.NaN; }  // same side of axis
    while (true) {
        double mid = (low+high)/2;
        double fMid = function.apply(mid);
        if (Math.abs(fMid)<THETA) {return mid;}
        else if (Math.signum(fLow) == Math.signum(fMid) ){  low = mid; fLow=fMid; }
        else {  high = mid;  fHigh=fMid; }
    }
}
```

Partially Ordered Trees

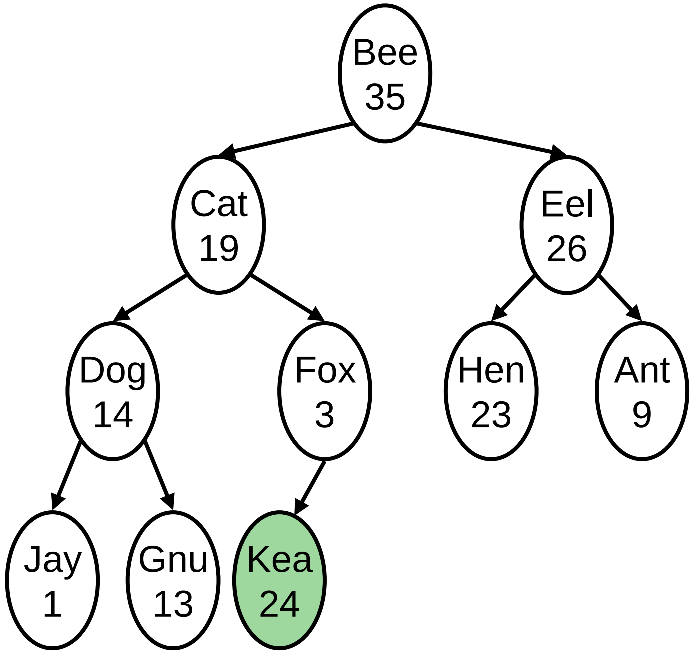
- **Partially Ordered Tree - implementing Priority Queues efficiently**

- Binary tree
- Children ≤ parent,
- Order of children not important



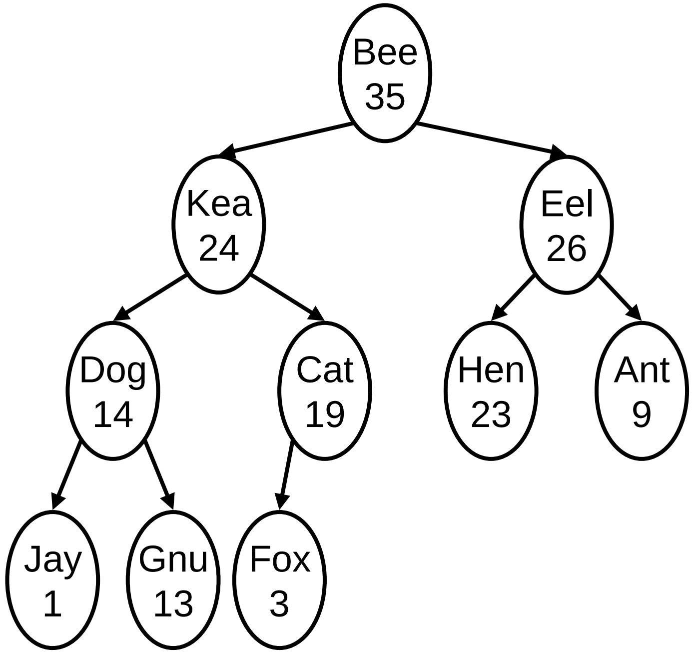
Partially Ordered Tree: add

- 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!

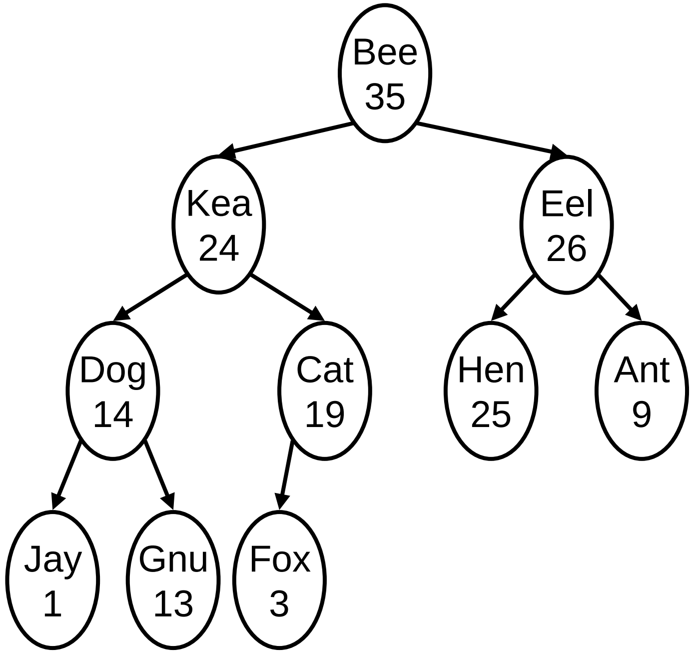


Partially Ordered Tree: remove I

- 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!

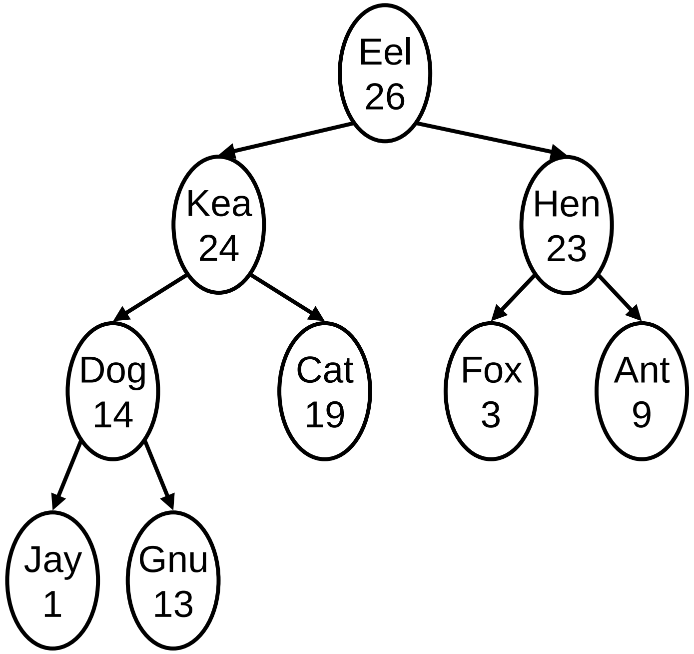


Partially Ordered Tree: remove II

- 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!

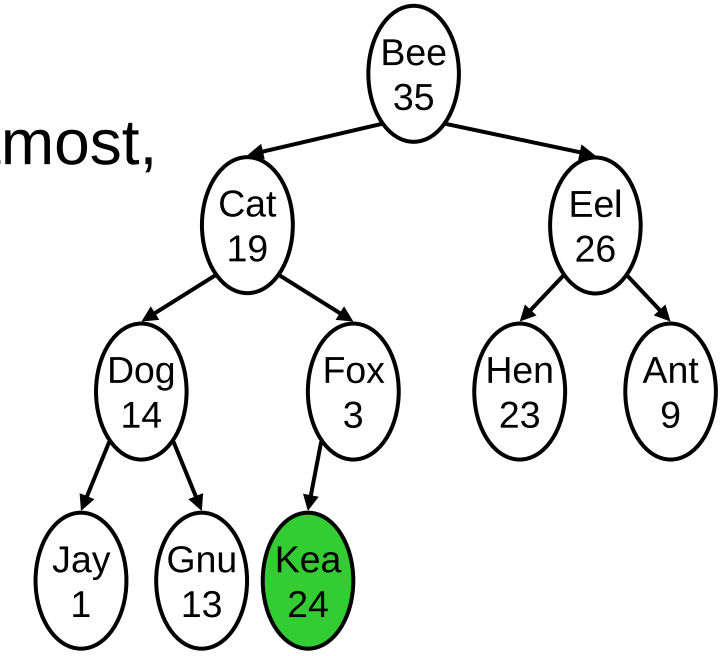
Alternative:

- 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:

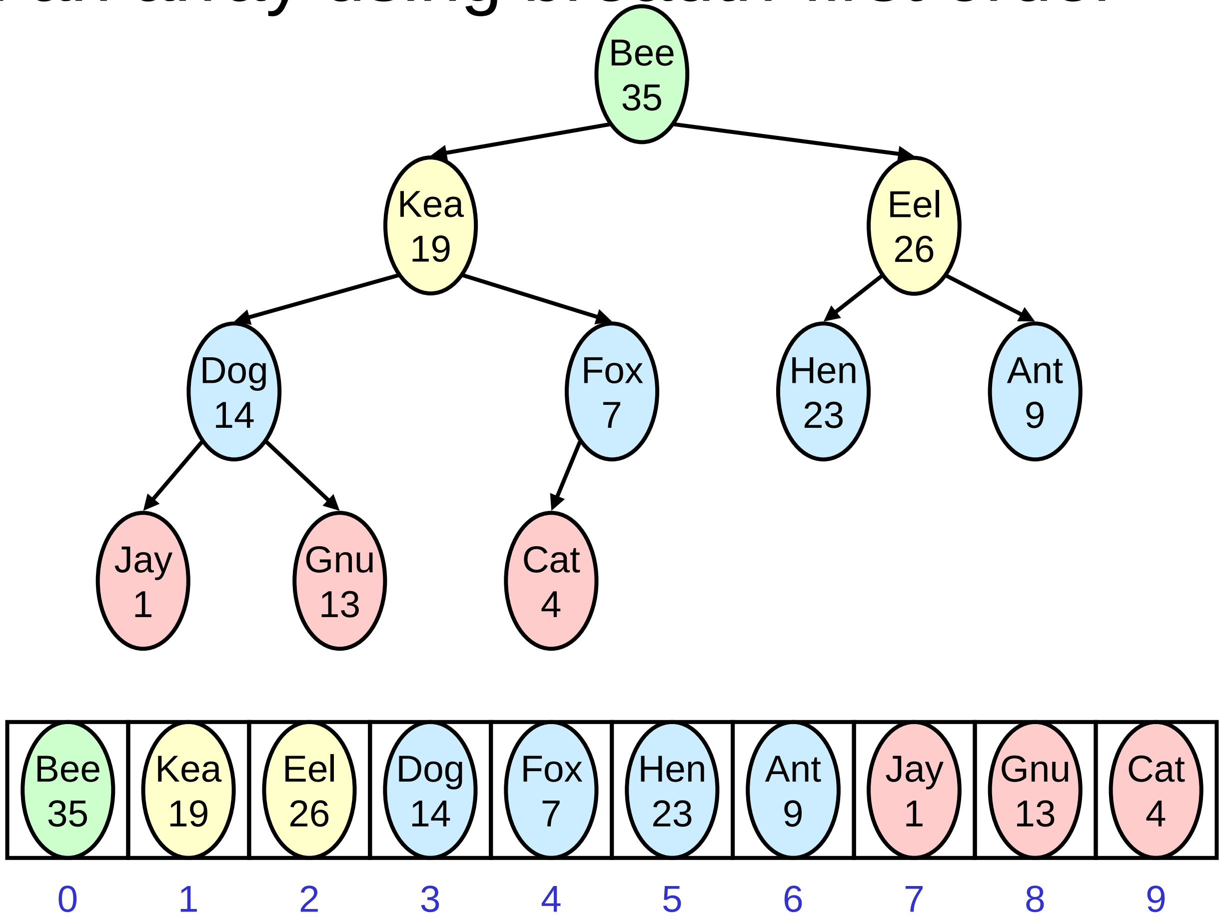
- the bottom right node,
- children from parent,
- parent from children.

Heap:

- A complete, partially ordered, binary tree

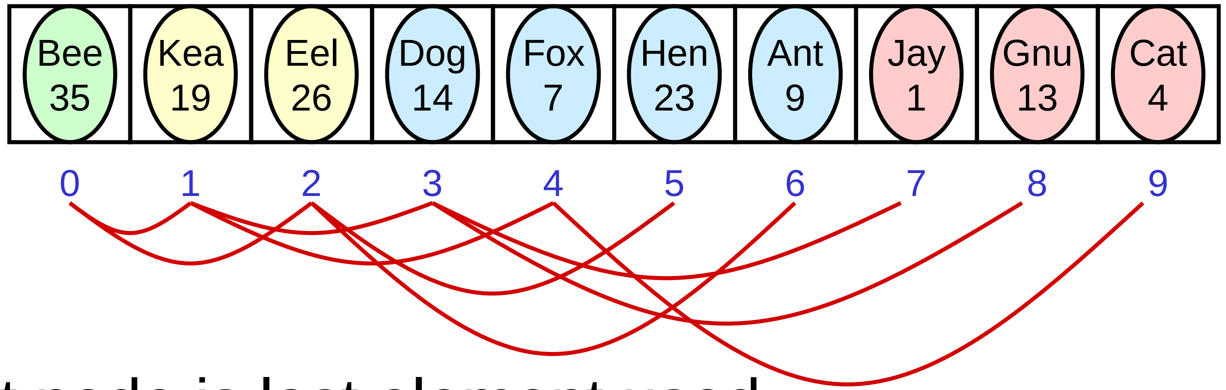
  complete = every level full, except bottom, where nodes are to the left

- Implemented in an array using breadth-first order



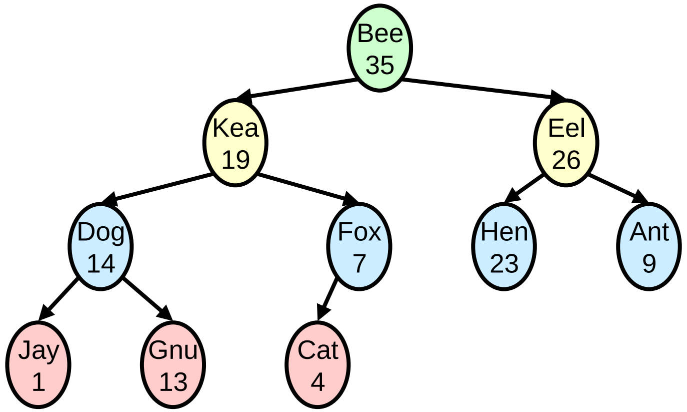
Heap

- 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

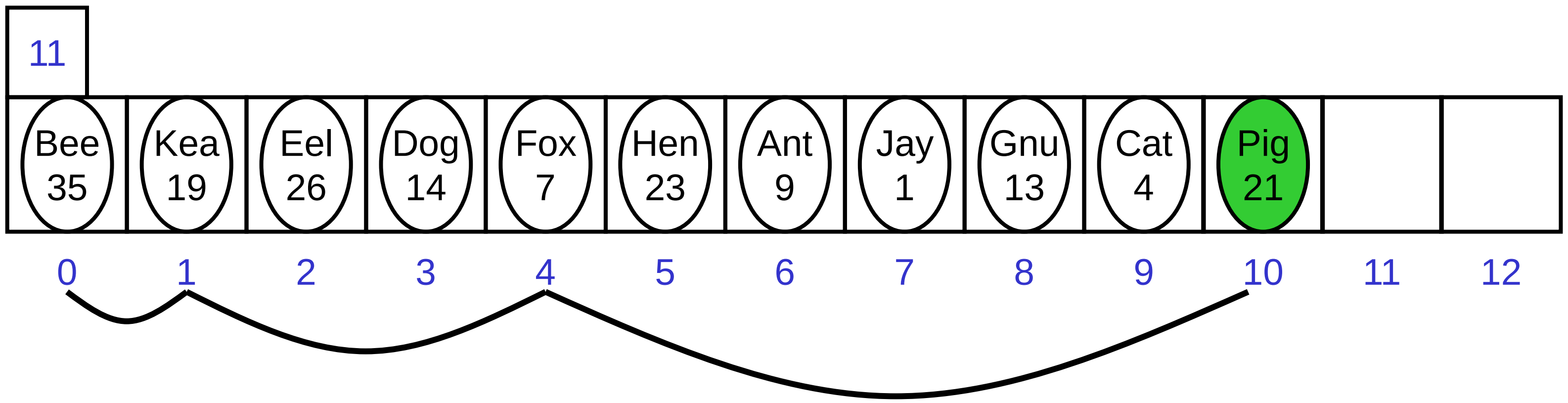


- Bottom right node is last element used.

- There are no gaps!



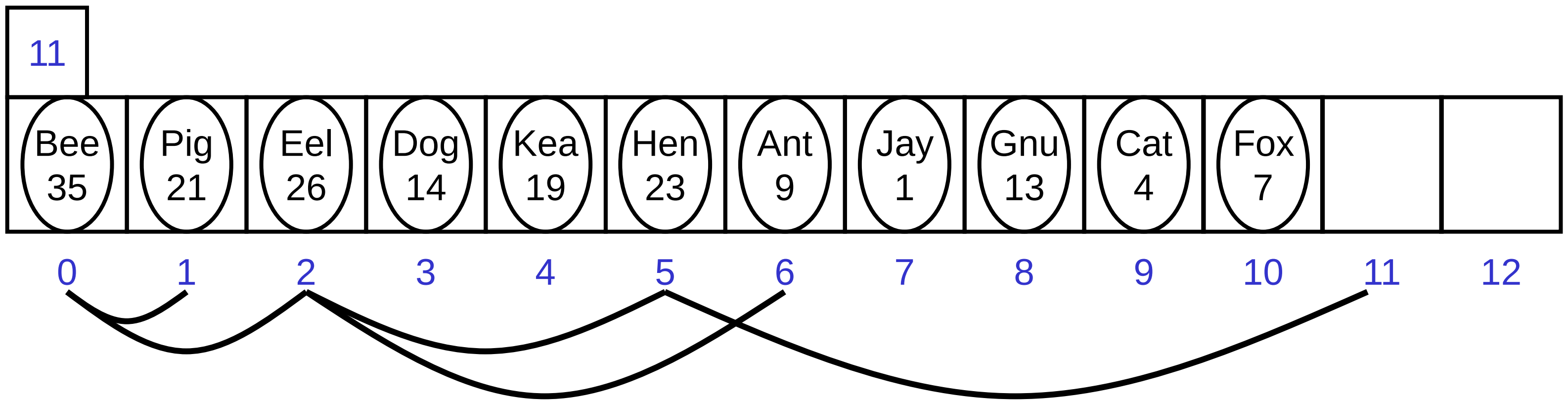
Heap: add



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 2×0+1 = 1, 2×0+2 = **2**
  - If smaller than largest child, swap
- Find largest child 2×2+1 = 5, 2×2+2 = 6
  - If smaller than largest child, swap
- Find largest child 2×5+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;
    }
}

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;
    }
}
```

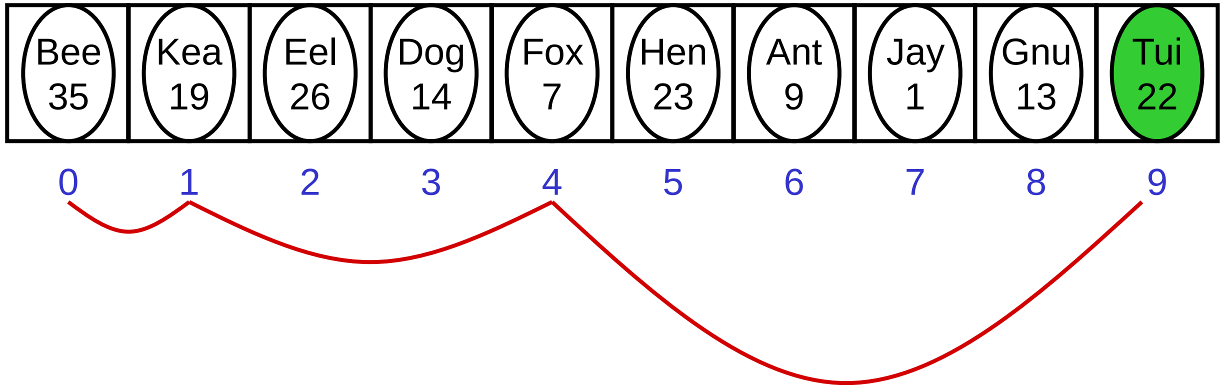
add at the end of the array

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);
    }
}
```

recurse up the tree...

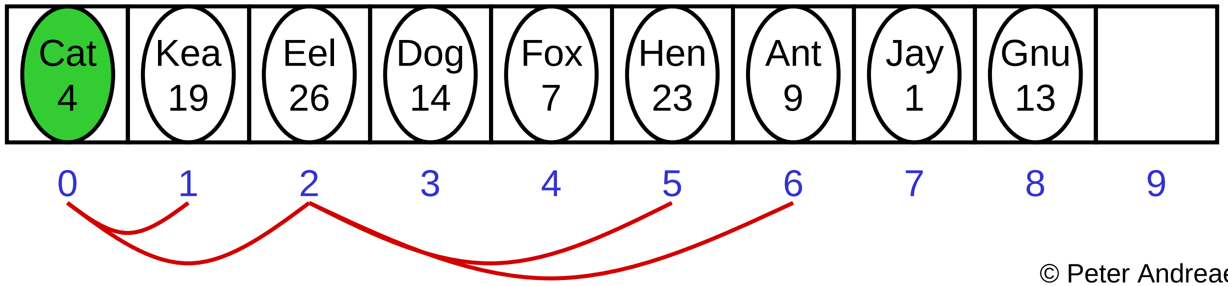



```
private void swap(List<E> data, int from, int to)
  data.set(child, data.set(parent, data.get(child)));
```

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!!