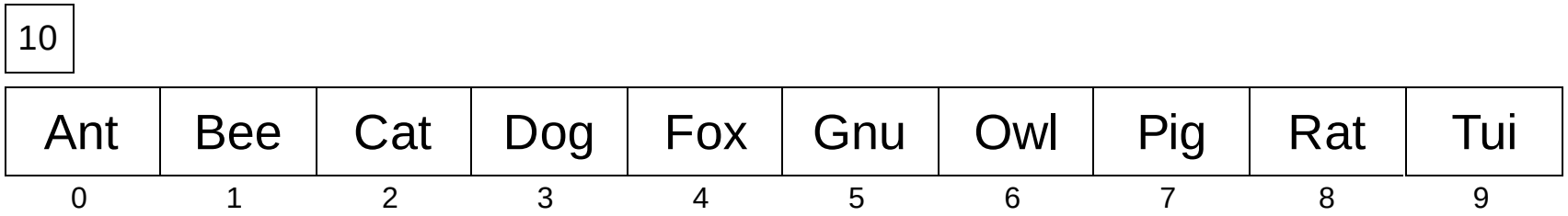
Searching for Items in a sorted List.

- 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”



  - 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
```

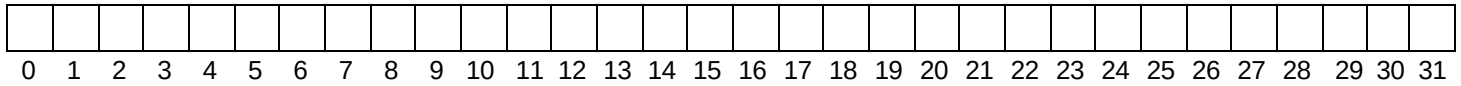
Another form of Binary Search

```
/* 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

  1 n

  2

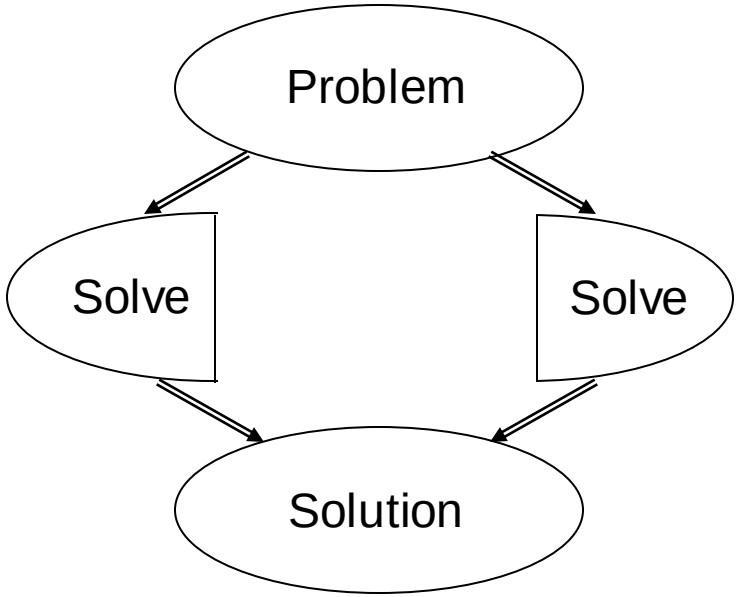
  k 1

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

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

$\lg(1000) \approx 10$, $\lg(1,000,000) \approx 20$, $\lg(1,000,000,000) \approx 30$

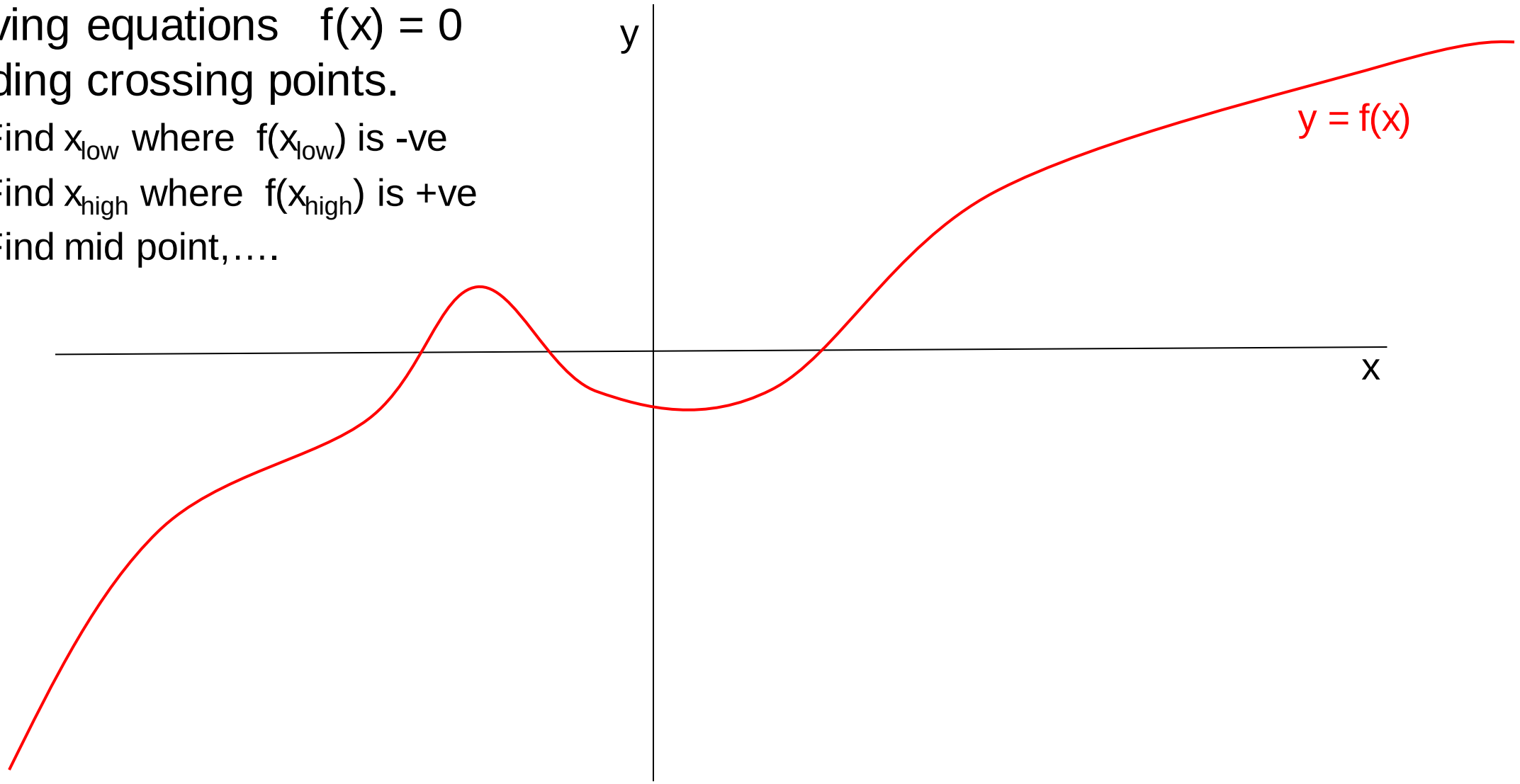
Every time you double n, you add one $lg(n)$



- 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,….



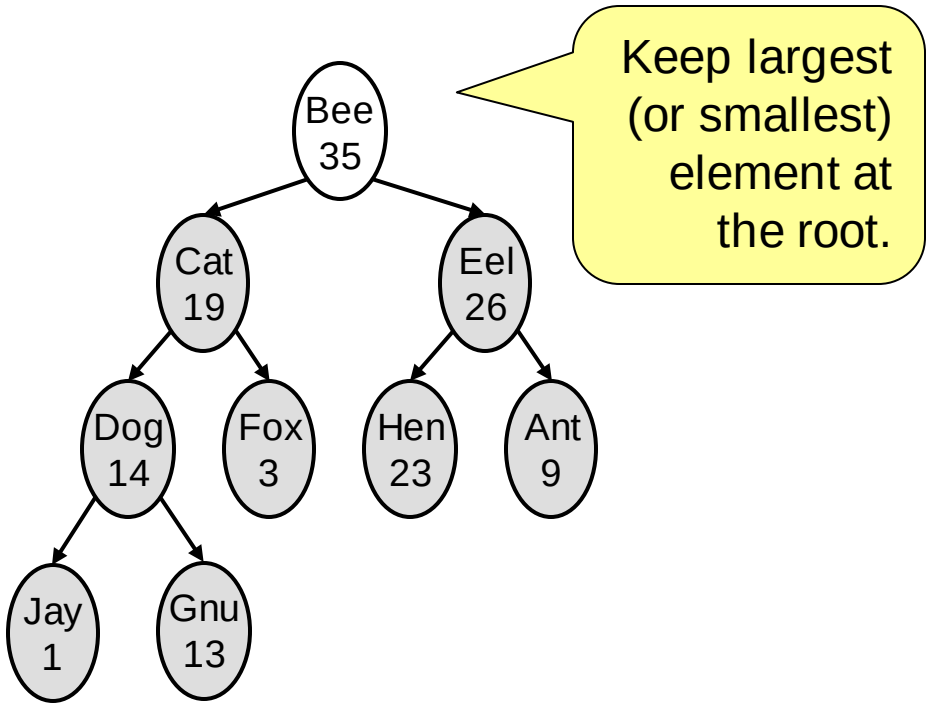
Bisection

```
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; }
    }
}
```

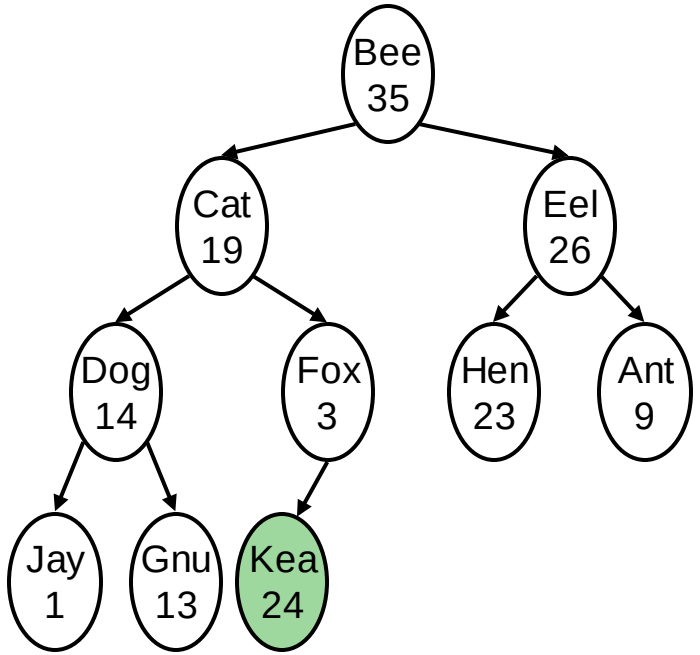
Partially Ordered Trees

- **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!