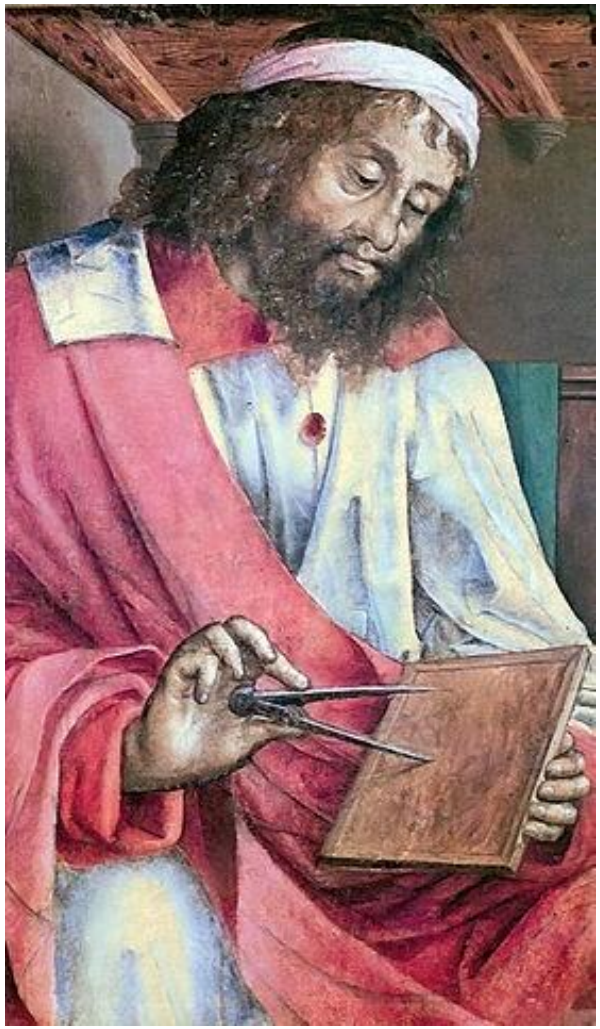
Video 1.1
Sampath Kannan

What is an algorithm?

Muhammad ibn Musa al-Khwarizmi: gave rise to the word "algorithm"



Euclid: Inventor of an algorithm for computing greatest common divisors

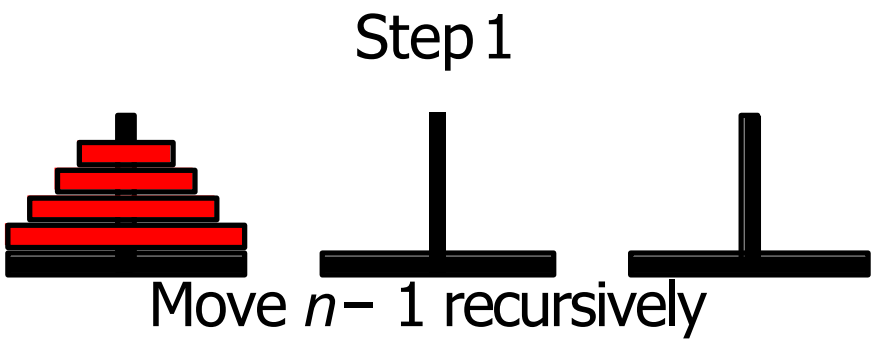
Why study algorithms?

As programs get complicated, thinking algorithmically allows us to:

- reason about their correctness and efficiency before implementing them
- focus on techniques for solving problems
- understand relationship between different computational problems

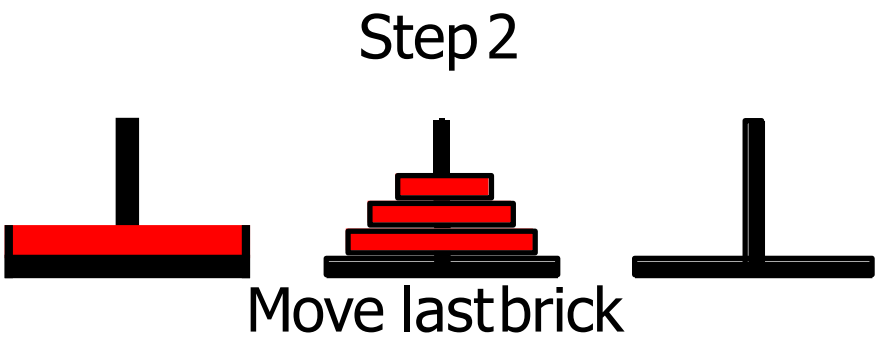
Induction + Algorithm Design

- A fundamental idea in algorithm design – solve a problem on bigger data sets using your knowledge of how to solve it on smaller ones.
- This idea embodies the proof technique of Mathematical Induction.
- Example: Towers of Hanoi



Induction + Algorithm Design

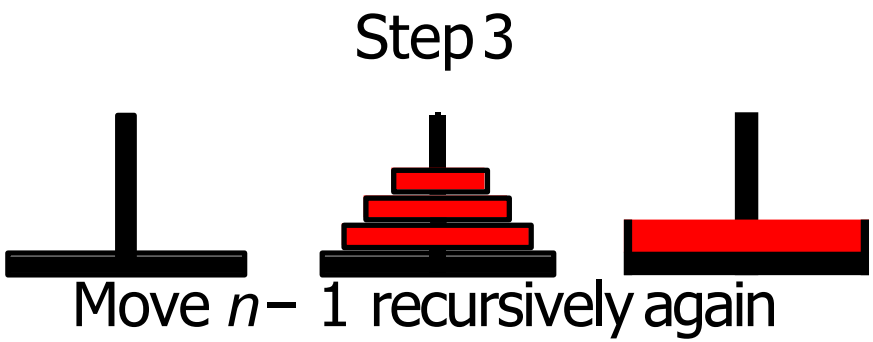
- A fundamental idea in algorithm design – solve a problem on bigger data sets using your knowledge of how to solve it on smaller ones.
- This idea embodies the proof technique of Mathematical Induction.
- Example: Towers of Hanoi



- Move top n-1 disks from rod A to rod B

Induction + Algorithm Design

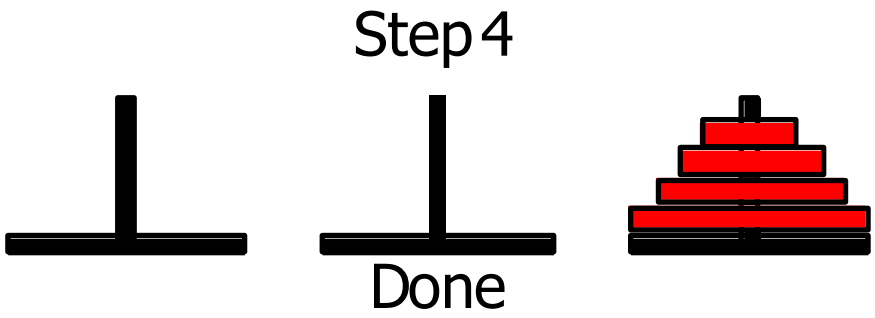
- A fundamental idea in algorithm design – solve a problem on bigger data sets using your knowledge of how to solve it on smaller ones.
- This idea embodies the proof technique of Mathematical Induction.
- Example: Towers of Hanoi



- Move top n-1 disks from rod A to rod B
- Move disk 1 from rod A to rod C

Induction + Algorithm Design

- A fundamental idea in algorithm design – solve a problem on bigger data sets using your knowledge of how to solve it on smaller ones.
- This idea embodies the proof technique of Mathematical Induction.
- Example: Towers of Hanoi



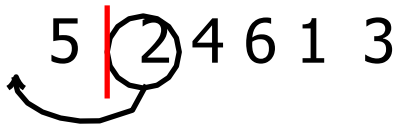
- Move top n-1 disks from rod A to rod B
- Move disk 1 from rod A to rod C
- Move the n-1 disks from rod B to rod C

Induction + Algorithm Design

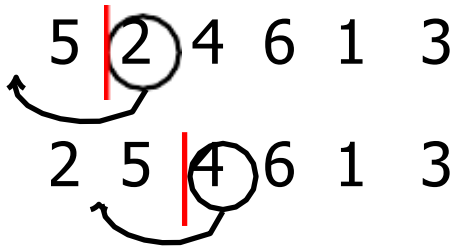
- A fundamental idea in algorithm design – solve a problem on bigger data sets using your knowledge of how to solve it on smaller ones.
- This idea embodies the proof technique of Mathematical Induction.
- Example: Towers of Hanoi

- Move top n-1 disks from rod A to rod B
- Move disk 1 from rod A to rod C
- Move the n-1 disks from rod B to rod C
- How long does this take? How can this be analyzed with **induction**?

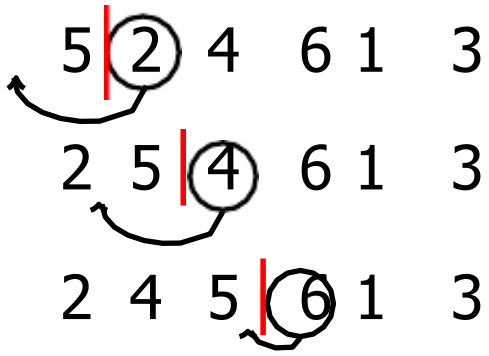
Another Example: Insertion Sort



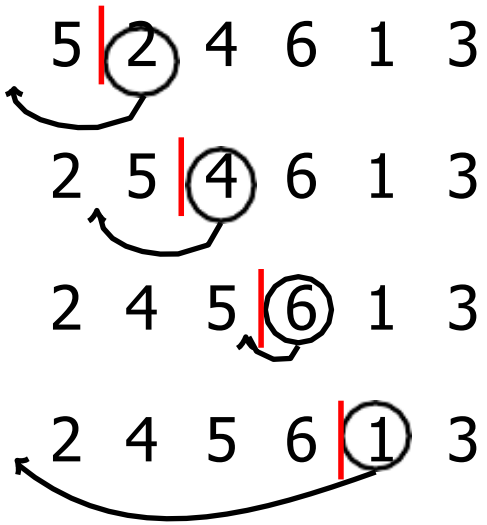
Another Example: Insertion Sort



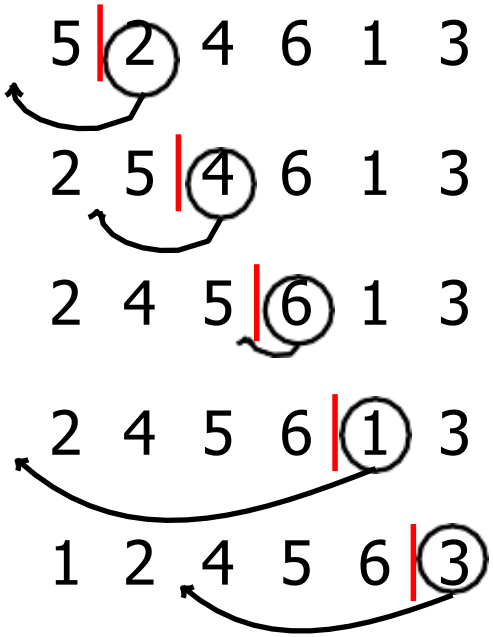
Another Example: Insertion Sort



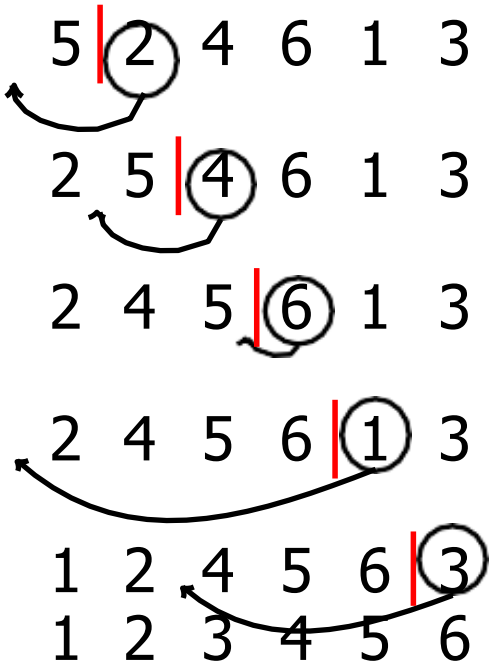
Another Example: Insertion Sort



Another Example: Insertion Sort

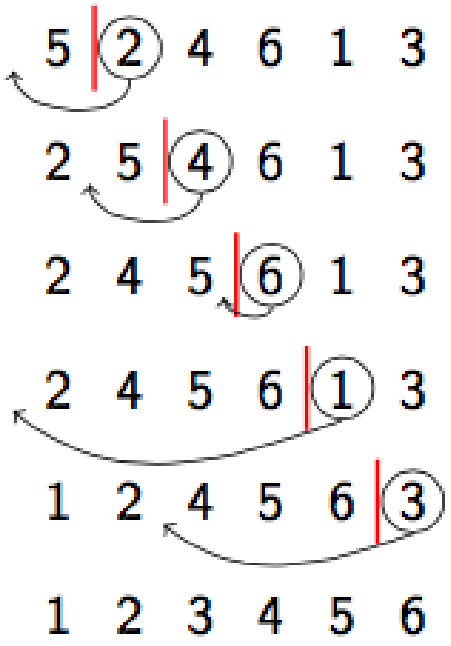


Another Example: Insertion Sort



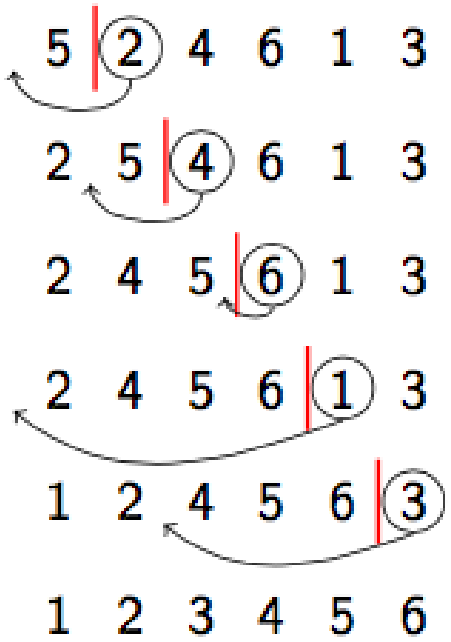
Another Example: Insertion Sort

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```



Another Example: Insertion Sort

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```



If we've already sorted the first **k elements** of the array, how long does it take to place the next element?

Recurrence Relations

- How can we analyze the runtime of an algorithm that is recursive?

Recurrence Relations

- How can we analyze the runtime of an algorithm that is recursive?
- Recurrence relation: a function defined in terms of itself

Recurrence Relations

- How can we analyze the runtime of an algorithm that is recursive?
- Recurrence relation: a function defined in terms of itself
- How can we write the runtime of Towers of Hanoi using a recurrence?

Recurrence Relations

- $T(n)$ = # operations required to solve a tower with n disks

Recurrence Relations

- $T(n) =$ # operations required to solve a tower with n disks
- $T(n-1) =$ # operations required to solve a tower with $n-1$ disks

Recurrence Relations

- $T(n) =$ # operations required to solve a tower with n disks
- $T(n-1) =$ # operations required to solve a tower with $n-1$ disks
- Can we write $T(n)$ using $T(n-1)$?

Recurrence Relations

Towers of Hanoi recurrence: $T(n) = 2T(n-1) + 1$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2T(n-1) + 1$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2T(n-1) + 1$
- $T(n-1) = 2T(n-2) + 1$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- substituting in for $T(n-1)$:

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping** substituting in for $T(n-1)$:

- $T(n) = 2(2T(n-2)+1)+1$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- substituting in for $T(n-1)$:
- $T(n) = 2(2T(n-2)+1) + 1$
- $T(n) = 4T(n-2)+2+1$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- substituting in again for $T(n-2)$:

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- substituting in again for $T(n-2)$:
- $T(n) = 8T(n-3) + 4 + 2 + 1$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- substituting in again for $T(n-2)$:
- $T(n) = 8T(n-3) + 4 + 2 + 1$
- Can we generalize this to k?

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2^k T(n-k) + (\sum_{i=0}^{k-1} 2^i)$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2^k T(n-k) + (\sum_{i=0}^{k-1} 2^i)$
- $T(n) = 2^k T(n-k) + (2^k - 1)$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2^k T(n-k) + (\sum_{i=0}^{k-1} 2^i)$
- $T(n) = 2^k T(n-k) + (2^k - 1)$
- What is $T(1)$?

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2^k T(n-k) + (\sum_{i=0}^{k-1} 2^i)$
- $T(n) = 2^k T(n-k) + (2^k - 1)$
- What is $T(1)$?
  - How long does it take to solve a tower with 1 ring?

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2^k T(n-k) + (\sum_{i=0}^{k-1} 2^i)$
- $T(n) = 2^k T(n-k) + (2^k - 1)$
- What is $T(1)$?
  - How long does it take to solve a tower with 1 ring?
  - $T(1) = 1$. Substitute $k = n - 1$
- $T(n) = 2^{n-1} + 2^{n-1} - 1$

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2^k T(n-k) + (\sum_{i=0}^{k-1} 2^i)$
- $T(n) = 2^k T(n-k) + (2^k - 1)$
- What is $T(1)$?
  - How long does it take to solve a tower with 1 ring?
  - $T(1) = 1$. Substitute $k = n - 1$
- $T(n) = 2^{n-1} + 2^{n-1} - 1$
- $T(n) = 2^n - 1$

Towers of Hanoi: Runtime

Result: Solving Towers of Hanoi requires $2^n - 1$ operations!

Towers of Hanoi: Runtime

We can expand this recurrence out through **telescoping**

- $T(n) = 2T(n-1) + 1$
- $T(n-1) = 2T(n-2) + 1$
- substituting in for $T(n-1)$:
- $T(n) = 2(2T(n-2) + 1) + 1$
- $T(n) = 4T(n-2) + 2 + 1$
- substituting in again for $T(n-2)$:
- $T(n) = 8T(n-3) + 4 + 2 + 1$
- Can we generalize this to k?

- $T(n) = 2^k T(n-k) + (\sum_{i=0}^{k-1} 2^i)$
- $T(n) = 2^k T(n-k) + (2^k - 1)$
- What is $T(1)$?
  - How long does it take to solve a tower with 1 ring?
  - $T(1) = 1$. Substitute $k = n - 1$
- $T(n) = 2^{n-1} + 2^{n-1} - 1$
- $T(n) = 2^n - 1$

Result: Solving Towers of Hanoi requires $2^n - 1$ operations!

Recurrence Relations: Back to Insertion Sort

- Can we write InsertionSort using a recurrence?

Recurrence Relations: Back to Insertion Sort

- Can we write InsertionSort using a recurrence?
- Not really, it isn't recursive! Instead, we can analyze how long each iteration of the loop takes.

Recurrence Relations: Back to Insertion Sort

- Can we write InsertionSort using a recurrence?
- Not really, it isn't recursive! Instead, we can analyze how long each iteration of the loop takes.
- Key observation: At the kth iteration of the loop, the first $k - 1$ elements of the array are in sorted order

Recurrence Relations: Back to Insertion Sort

- Can we write InsertionSort using a recurrence?
- Not really, it isn't recursive! Instead, we can analyze how long each iteration of the loop takes.
- Key observation: At the kth iteration of the loop, the first $k - 1$ elements of the array are in sorted order
- First iteration of the loop: 0 swaps required (first element is trivially sorted)
- Last iteration of the loop: at most $n - 1$ swaps required

Recurrence Relations: Back to Insertion Sort

- Can we write InsertionSort using a recurrence?
- Not really, it isn't recursive! Instead, we can analyze how long each iteration of the loop takes.
- Key observation: At the kth iteration of the loop, the first $k - 1$ elements of the array are in sorted order
- First iteration of the loop: 0 swaps required (first element is trivially sorted)
- Last iteration of the loop: at most $n - 1$ swaps required
- In general, kth iteration of the loop: at most $k - 1$ swaps required

Recurrence Relations: Back to Insertion Sort

Finding the total number of swaps:

- total number of swaps = $\sum_{i=0}^{n} i - 1$

Recurrence Relations: Back to Insertion Sort

Finding the total number of swaps:

- total number of swaps = $\sum_{i=0}^{n} i - 1$
- $= 1 + 2 + \ldots + n - 1$
- $= \frac{n(n-1)}{2}$

Recurrence Relations: Back to Insertion Sort

Finding the total number of swaps:

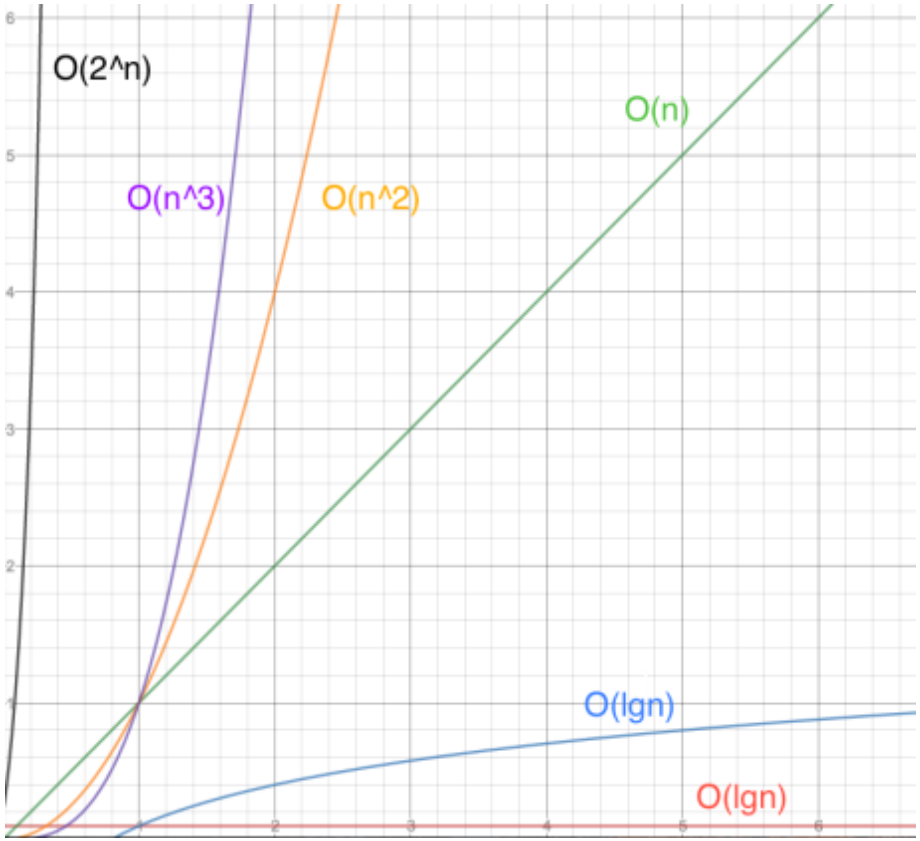
- total number of swaps = $\sum_{i=0}^{n} i - 1$
- $= 1 + 2 + \ldots + n - 1$
- $= \frac{n(n-1)}{2}$

Number of swaps required for insertion sort: $\frac{n(n-1)}{2}$

Video 1.2
Sampath Kannan

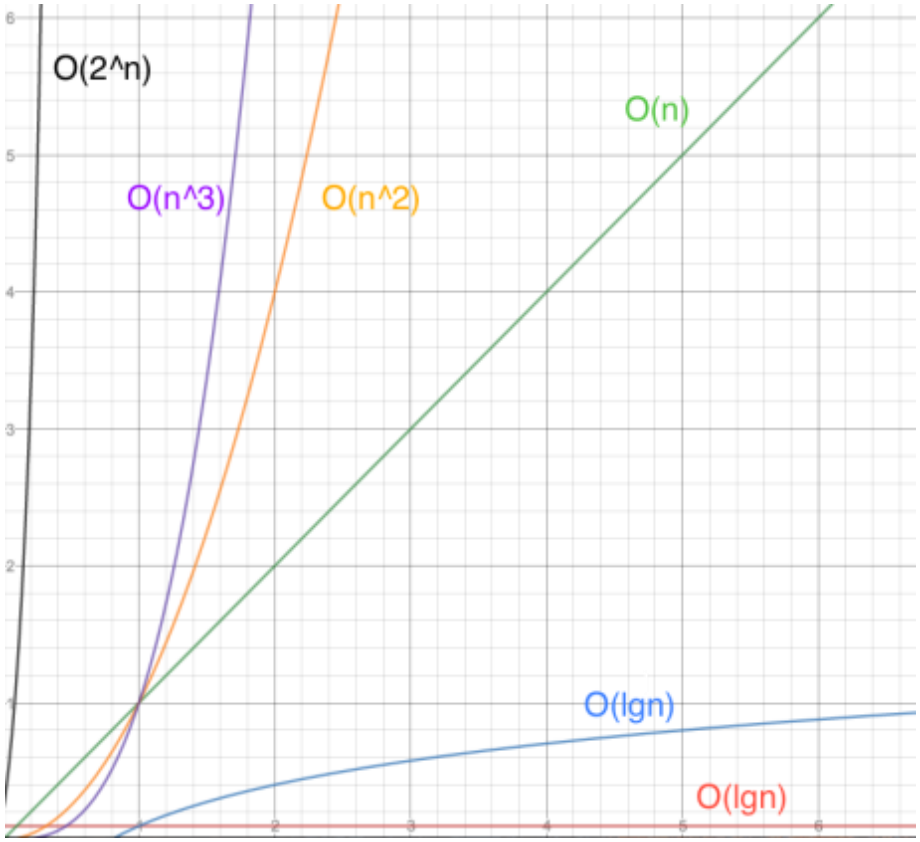
Asymptotic Bounds

Motivation:



Asymptotic Bounds

Motivation:



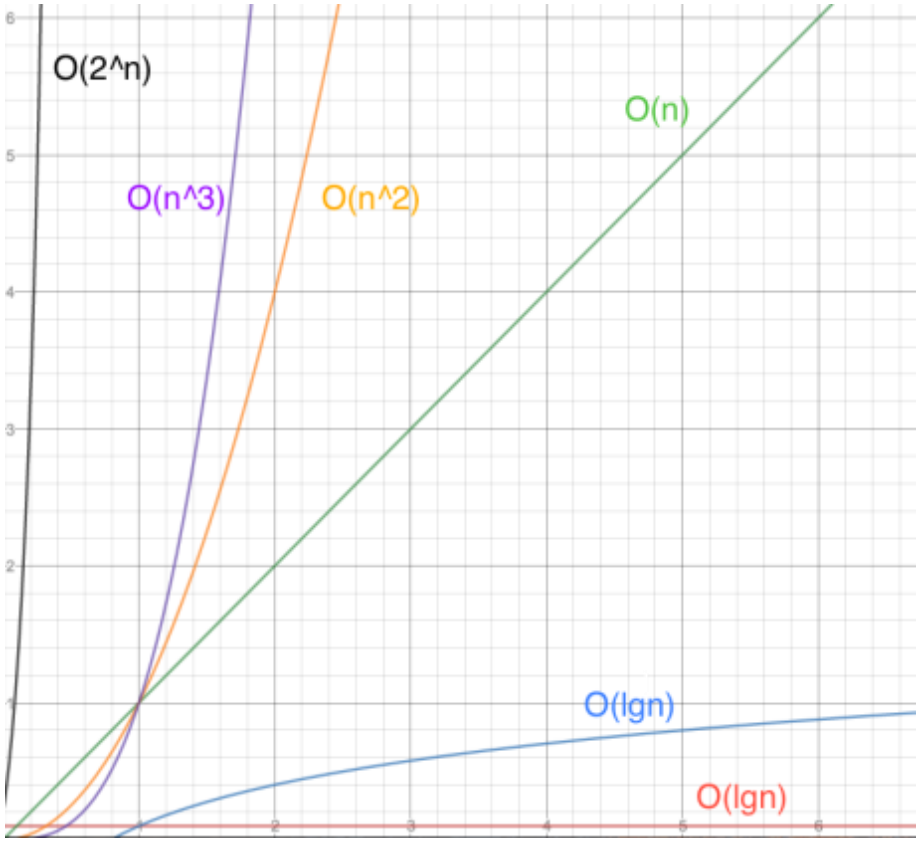
- Essentially a way to compare functions without worrying about their behavior on small n.

  Big-Oh is like $\leq$ (ignoring constant factors), and Big-Omega is like $\geq$

- Gives us an idea of how fast a function grows

Asymptotic Bounds

Motivation:



- Essentially a way to compare functions without worrying about their behavior on small n. In this sense Big-Oh is like $\leq$ (ignoring constant factors), and Big-Omega is like $\geq$
- Gives us an idea of how fast a function grows

- Note: $O(f(n))$ is a **set**.
  - $O(n^2)$: the set of all function that do not grow faster than n^2

Asymptotic Bounds: Examples

Some elements of $O(n^2)$:

- $2n^2 \in O(n^2)$
- $100n^2 + n + 1 \in O(n^2)$
- $n \in O(n^2)$

Some elements of $\Omega(n^2)$:

- $2n^2 \in \Omega(n^2)$
- $\frac{n^2}{1000} n \in \Omega(n^2)$
- $n^3 \in \Omega(n^2)$

What is the complexity of insertion sort?

- $T(n) = T(n-1) + n$
- $T(n) = \frac{n^2+n}{2}$
- $T(n) \in O(n^2)$

Insertion sort has a runtime of $O(n^2)$

Complexity

More on Insertion Sort...

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```

Complexity

More on Insertion Sort...

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```

How does insertion sort perform on an already-sorted array?

Complexity

More on Insertion Sort...

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```

How does insertion sort perform on an already-sorted array?

Complexity

More on Insertion Sort...

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```

How does insertion sort perform on an already-sorted array?

time for inner loop?

Complexity

More on Insertion Sort...

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```

How does insertion sort perform on an already-sorted array?

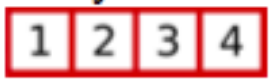
- time for inner loop?
- Each iteration requires no swaps! (constant time to check the first element)
- $\sum_{i=1}^{n} 1 = n \in O(n)$

Complexity

More on Insertion Sort...

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```

How does insertion sort perform on an already-sorted array?



- time for inner loop?
- Each iteration requires no swaps! (constant time to check the first element)

- $\sum_{i=1}^{n} 1 = n \in O(n)$
- is insertion sort $O(n)$, or $O(n^2)$?

Complexity

More on Insertion Sort…

```
insertion-sort A:
    for i <- 1 to length(A)
        j <- i
        while j > 0 and A[j-1] > A[j]:
            swap A[j] and A[j-1]
            j <- j-1
```

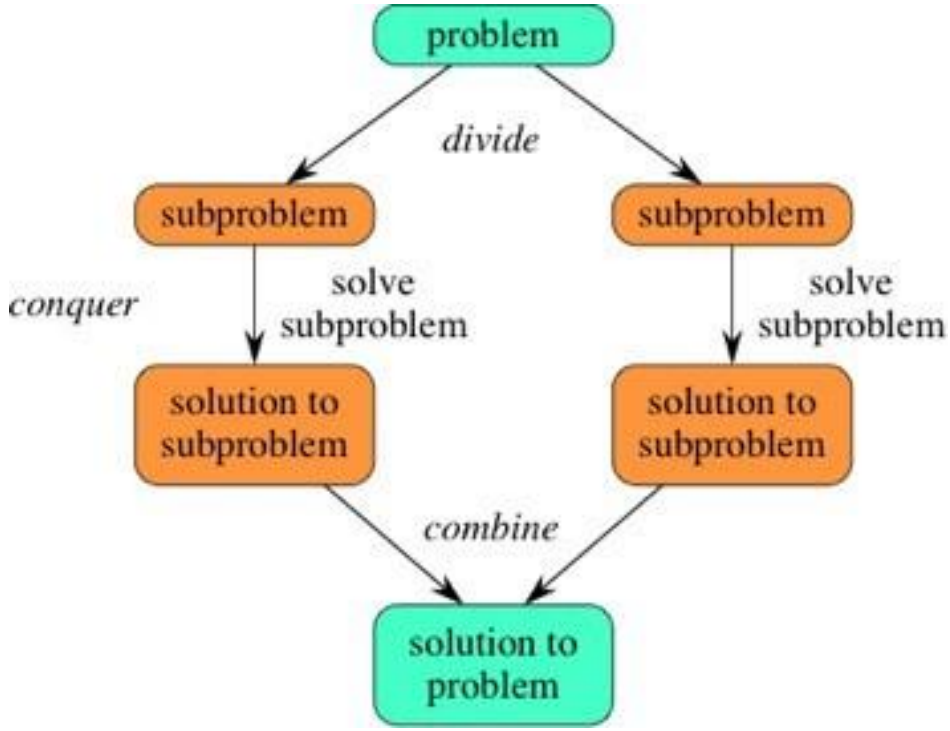
How does insertion sort perform on an already-sorted array?

- time for inner loop?
- Each iteration requires no swaps! (constant time to check the first element)
- $\sum_{i=1}^{n} 1 = n \in O(n)$
- is insertion sort $O(n)$, or $O(n^2)$?

Takeaway: Can't assume anything about the input. Always assume the worst case!

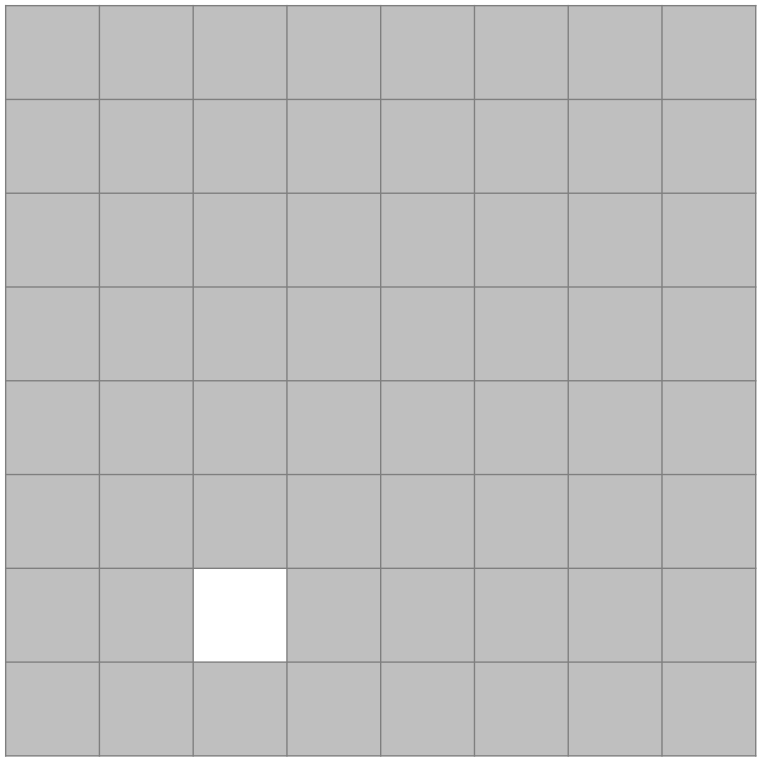
Algorithm Design: Divide and Conquer Paradigm

Idea: Solve a problem by splitting it into pieces, solving those pieces recursively, and merging them to solve the larger problem



Divide and Conquer Example: Triominos

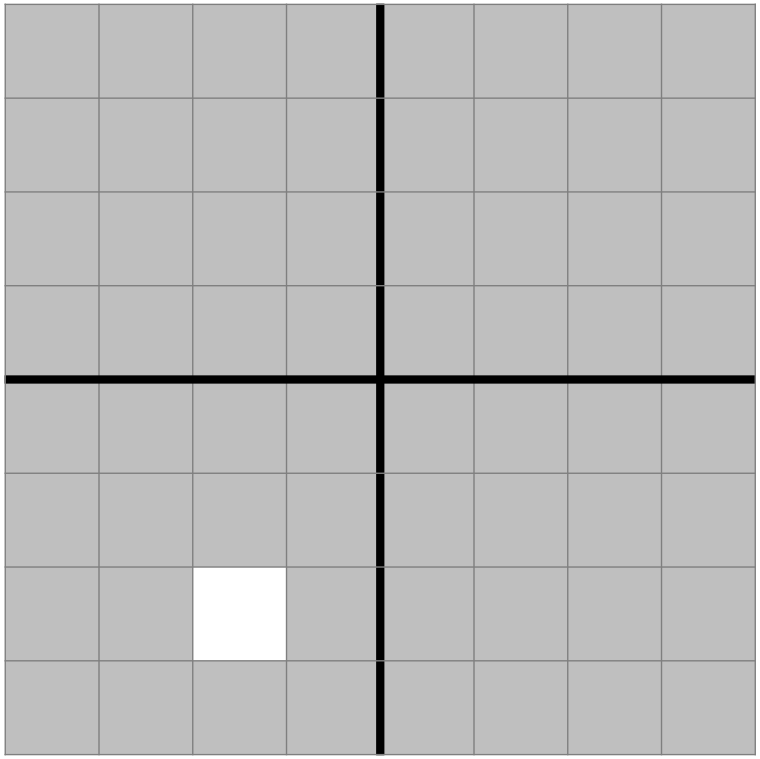
- Input: *NxN* grid (assume N is a power of 2) with a single square removed, and a supply of corner shaped triomino tiles
- Goal: Fill the grid without any overlapping tiles



Algorithm:

Divide and Conquer Example: Triominos

- Input: *NxN* grid (assume n is a power of 2) with a single square removed, and a supply of corner shaped triomino tiles
- Goal: Fill the grid without any overlapping tiles



Algorithm:

- **Divide** the grid into 4 squares (size $2^{n-1} \times 2^{n-1}$).
- note: 1 of these 4 squares contains the missing piece

Video 1.3
Sampath Kannan

Binary Search

- How long does it take to search for an element in an array? O(n)
- Idea: Can we do better if we know that the array is sorted?

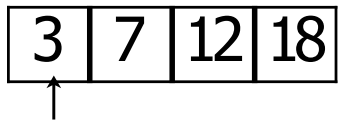
```
Binary-search(A, val):
    return Binary-search(A, val, 0, (length(A)-1))
```

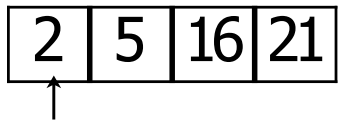
```
Binary-search(A, val, low, high): if
    high < low
        return -1 (not found)
    mid <- (low + high) / 2  if
    A[mid] > val
        return Binary-search(A, val, low, mid-1)
    else if A[mid] < val
        return Binary-search(A, val, mid+1, hi)
    else return mid
```

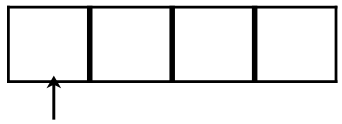
- Each step of the algorithm, the size of the input halves.
- $T(n) = T(\frac{n}{2}) + 1$
- How to solve this recurrence: How many times can we halve N before reaching 1? $\frac{N}{2}$, $\frac{N}{4}$, ...
- $\frac{N}{2^k} = 1, k = lg_2 N$
- binary search runs in $O(lgN)$

Merging two sorted lists

Input: two sorted arrays of size n and m Output: a single sorted array of size n+m





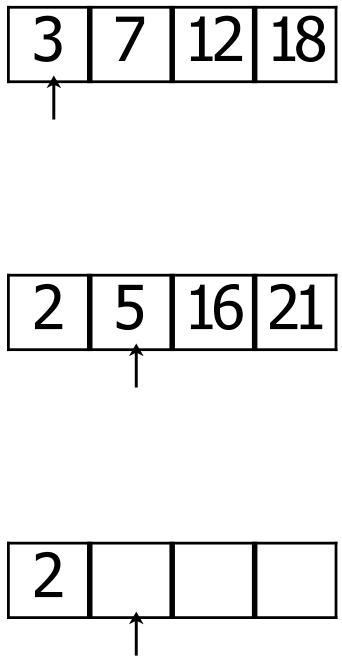


```
merge(A, B):
    C = new array[len(A) + len(B)]
    i, j, k <- 0
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            C[k] <- A[i]
            i++, k++
        else:
            C[k] <- B[j]
            j++, k++
    while i < len(A):
        C[k++] <- A[i++]
    while j < len(B):
        C[k++] <- B[j++]
    return C
```

- How long does this take?
- Every time a comparison is made, either i or j is incremented
- Total number of comparisons is n + m
- merging runs in $O(n + m)$ time

Merging two sorted lists

Input: two sorted arrays of size n and m Output: a single sorted array of size n+m

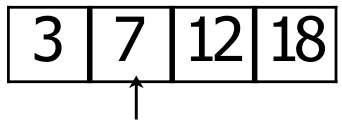


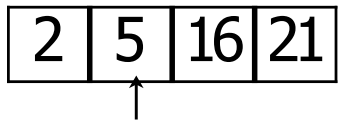

```
merge(A, B):
    C = new array[len(A) + len(B)]
    i, j, k <- 0
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            C[k] <- A[i]
            i++, k++
        else:
            C[k] <- B[j]
            j++, k++
    while i < len(A):
        C[k++] <- A[i++]
    while j < len(B):
        C[k++] <- B[j++]
    return C
```

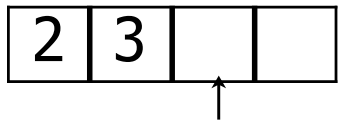
- How long does this take?
- Every time a comparison is made, either i or j is incremented
- Total number of comparisons is n + m
- merging runs in $O(n + m)$ time

Merging two sorted lists

Input: two sorted arrays of size n and m Output: a single sorted array of size n+m





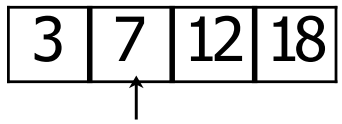


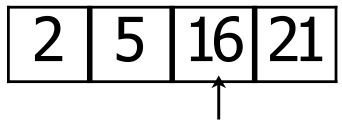

```
merge(A, B):
    C = new array[len(A) + len(B)]
    i, j, k <- 0
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            C[k] <- A[i]
            i++, k++
        else:
            C[k] <- B[j]
            j++, k++
    while i < len(A):
        C[k++] <- A[i++]
    while j < len(B):
        C[k++] <- B[j++]
    return C
```

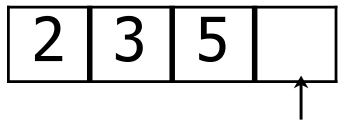
- How long does this take?
- Every time a comparison is made, either i or j is incremented
- Total number of comparisons is n + m
- merging runs in $O(n + m)$ time

Merging two sorted lists

Input: two sorted arrays of size n and m Output: a single sorted array of size n+m








```
merge(A, B):
    C = new array[len(A) + len(B)]
    i, j, k <- 0
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            C[k] <- A[i]
            i++, k++
        else:
            C[k] <- B[j]
            j++, k++
    while i < len(A):
        C[k++] <- A[i++]
    while j < len(B):
        C[k++] <- B[j++]
    return C
```

- How long does this take?
- Every time a comparison is made, either i or j is incremented
- Total number of comparisons is n + m
- merging runs in $O(n + m)$ time

More on Divide and Conquer: Mergesort

Input: An array of size n, Output: A sorted array of size n

Can we apply the Divide and Conquer paradigm to sorting? Idea: Split the array, sort halves recursively, **merge** the result

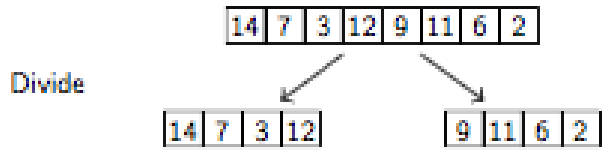

```
mergesort(A):
    mergesort(A, 0, len(A)-1)


mergesort(A, aux, lo, hi):
    if (hi - lo <= 1) return
    mid = (lo + hi) / 2
    mergesort(A, lo, mid)
    mergesort(A, mid+1, hi)
    C = merge(A[lo:mid], A[mid+1:hi])
    copy elements from C back into A
```

More on Divide and Conquer: Mergesort

Input: An array of size n, Output: A sorted array of size n

Can we apply the Divide and Conquer paradigm to sorting? Idea: Split the array, sort halves recursively, **merge** the result



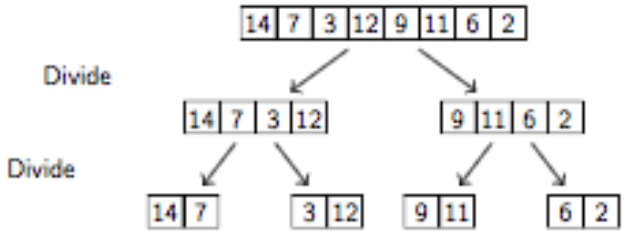

```
mergesort(A):
    mergesort(A, 0, len(A)-1)
```

```
mergesort(A, aux, lo, hi):
    if (hi - lo <= 1) return
    mid = (lo + hi) / 2
    mergesort(A, lo, mid)
    mergesort(A, mid+1, hi)
    C = merge(A[lo:mid], A[mid+1:hi])
    copy elements from C back into A
```

More on Divide and Conquer: Mergesort

Input: An array of size n, Output: A sorted array of size n

Can we apply the Divide and Conquer paradigm to sorting? Idea: Split the array, sort halves recursively, **merge** the result




```
mergesort(A):
    mergesort(A, 0, len(A)-1)
```

```
mergesort(A, aux, lo, hi):
    if (hi - lo <= 1) return
    mid = (lo + hi) / 2
    mergesort(A, lo, mid)
    mergesort(A, mid+1, hi)
    C = merge(A[lo:mid], A[mid+1:hi])
    copy elements from C back into A
```

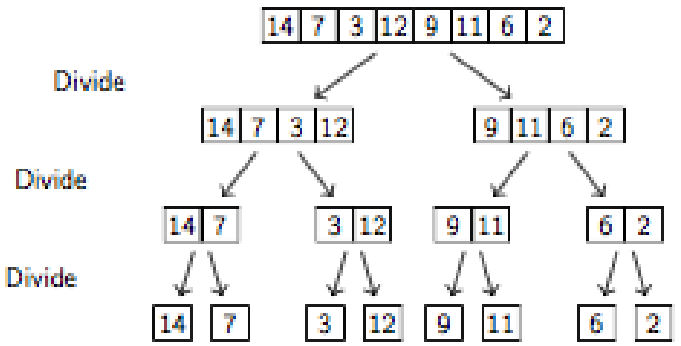
More on Divide and Conquer: Mergesort

Input: An array of size n, Output: A sorted array of size n

Can we apply the Divide and Conquer paradigm to sorting? Idea: Split the array, sort halves recursively, **merge** the result

```
mergesort(A):
    mergesort(A, 0, len(A)-1)


mergesort(A, aux, lo, hi):
    if (hi - lo <= 1) return
    mid = (lo + hi) / 2
    mergesort(A, lo, mid)
    mergesort(A, mid+1, hi)
    C = merge(A[lo:mid], A[mid+1:hi])
    copy elements from C back into A
```



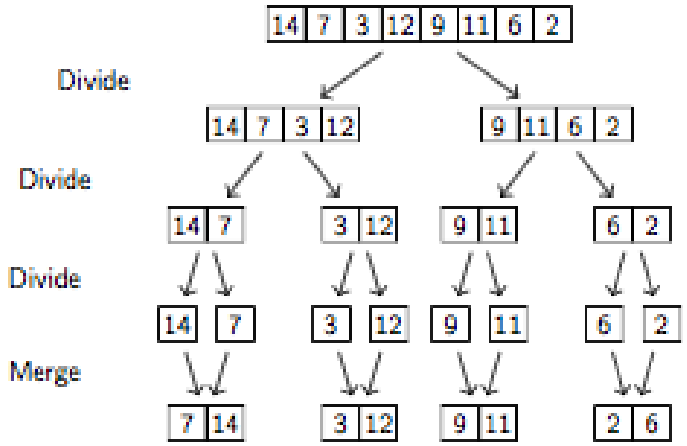
More on Divide and Conquer: Mergesort

Input: An array of size n, Output: A sorted array of size n

Can we apply the Divide and Conquer paradigm to sorting? Idea: Split the array, sort halves recursively, **merge** the result

```
mergesort(A):
    mergesort(A, 0, len(A)-1)


mergesort(A, aux, lo, hi):
    if (hi - lo <= 1) return
    mid = (lo + hi) / 2
    mergesort(A, lo, mid)
    mergesort(A, mid+1, hi)
    C = merge(A[lo:mid], A[mid+1:hi])
    copy elements from C back into A
```



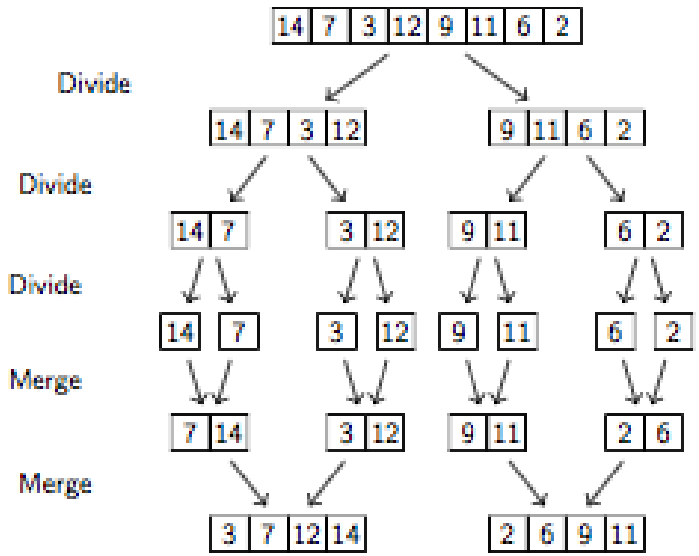
More on Divide and Conquer: Mergesort

Input: An array of size n, Output: A sorted array of size n

Can we apply the Divide and Conquer paradigm to sorting? Idea: Split the array, sort halves recursively, **merge** the result

```
mergesort(A):
    mergesort(A, 0, len(A)-1)
```

```
mergesort(A, aux, lo, hi):
    if (hi - lo <= 1) return
    mid = (lo + hi) / 2
    mergesort(A, lo, mid)
    mergesort(A, mid+1, hi)
    C = merge(A[lo:mid], A[mid+1:hi])
    copy elements from C back into A
```



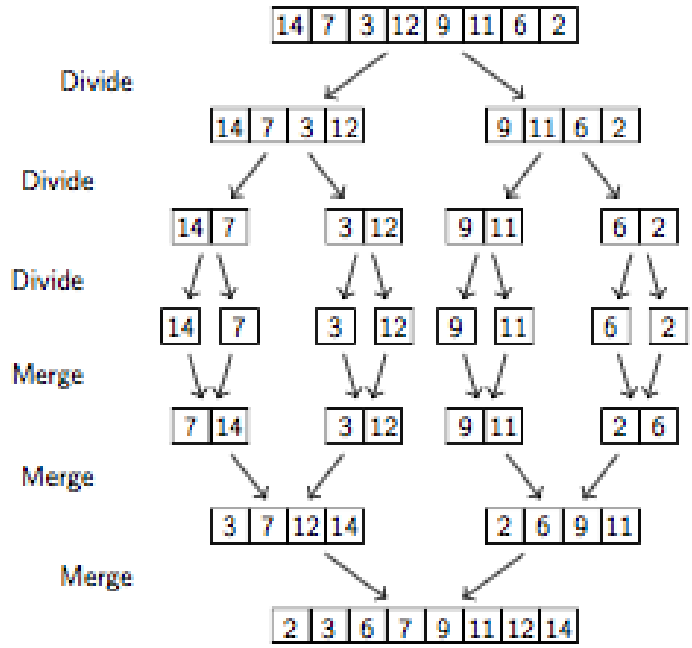
More on Divide and Conquer: Mergesort

Input: An array of size n, Output: A sorted array of size n

Can we apply the Divide and Conquer paradigm to sorting? Idea: Split the array, sort halves recursively, **merge** the result

```
mergesort(A):
    mergesort(A, 0, len(A)-1)
```

```
mergesort(A, aux, lo, hi):
    if (hi - lo <= 1) return
    mid = (lo + hi) / 2
    mergesort(A, lo, mid)
    mergesort(A, mid+1, hi)
    C = merge(A[lo:mid], A[mid+1:hi])
    copy elements from C back into A
```



Video 1.4
Sampath Kannan

Algorithm Design: Using Randomness

- Remember from InsertionSort: Algorithm performance can depend on the input:
  - on a sorted list: $O(n)$ comparisons
  - on a reversed list: $O(n^2)$ comparisons
  - In general: somewhere between n and $\frac{n(n+1)}{2}$ comparisons
  - However, the **worst-case** is still $O(n^2)$
- An "adversary" can repeatedly construct an input to our algorithm that causes it to perform as poorly as possible
- Can we prevent our algorithm performance from depending on the input?
  - Shift the dependency: from **input** to **randomization**
- Idea: Write algorithms that toss a coin!

First: An Introduction to Probability

- For a stronger introduction, see:
  https://www.coursera.org/learn/probability- intro

- *Random Variable*: A function X from the results of an experiment to numbers
- $E[X]$: the expected value of the random variable X (a "weighted average
  - Formula: $E[X] = \Sigma\, i * P(X = i)$ (for all values i that X can take on)
  - Example:
    - Roll a 6-sided die. Let X = the value that the die lands on. What is $E[X]$?
    - X can take on each of the values 1 through 6, each with probability $\frac{1}{6}$
    - $E[X] = 1\frac{1}{6} + 2\frac{1}{6} + .. + 6\frac{1}{6} = \frac{21}{6} = 3.5$

Intro to Probability: Continued

- What is the expected sum of two dice?
- X = the sum of two dice. Want to find $E[X]$.
- X can take on values from 2…12
- E.x. $P(X = 5)$. Can result from two die rolls of:
  - (1, 4)
  - (4, 1)
  - (2, 3)
  - (3, 2)
- $P(X = 5) = 4\frac{1}{36} = \frac{1}{9}$
- $E[X] = \sum_{i=2}^{12} i * P(X = i) = 2\frac{1}{36} + 3\frac{2}{36} + \ldots + 12\frac{1}{36}$

sum	2	3	4	5	6	7	8	9	10	11	12
probability	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Calculation is not trivial. Solution: Linearity of Expectation!

Intro to Probability: Continued

- *Linearity of Expectation*: For n random variables, $X_1, .., X_n$, $E[X_1 + .. + X_n] = E[X_1] + .. + E[X_n]$
- Example:
  - What is the expected sum of rolling 2 dice?
  - let X_i be the random variable denoting the value of the i'th die rolled
  - let X be the r.v. denoting the sum of all 2 dice
  - then $X = X_1 + X_2$
  - $E[X] = E[X_1 + X_2]$
  - $E[X] = E[X_1] + E[X_2]$ (by lin. of exp.)
  - as shown above, for each i, $E[X_i] = 3.5$
  - $E[X] = 3.5 + 3.5 = 7$

Expectation Example: Hat Checking

n people go to a restaurant, take off their hats and throw them in a pile. Afterwards, they each take a hat from the pile at random. What is the expected number of people who get their hat back?

- We can analyze using random variables!
- Let:
  - X = the number of people who get their hats back

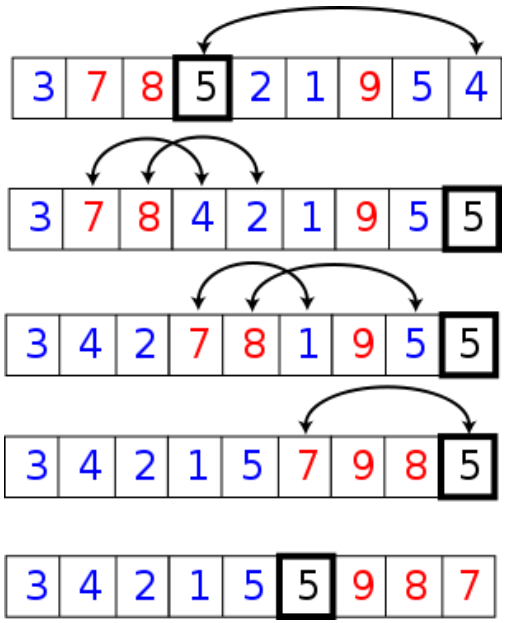
$$X_i = \begin{cases} 1 & \text{person } i \text{ chooses their own hat} \\ 0 & \text{person } i \text{ doesn't choose their own hat} \end{cases}$$

- What is $E[X_i]$?
  - From the definition:
  - $E[X_i] = 1 * P(\text{choose their hat}) + 0P(\text{don't choose their hat})$
  - $E[X_i] = P(\text{choose hat}) = \frac{1}{n}$
- Again, $X = X_1 + X_2 + .. + X_n$
- $E[X] = E[X_1 + .. + X_n] = E[X_1] + .. + E[X_n]$ by lin. of exp.
- $E[X] = n\frac{1}{n} = 1$

In expectation, one person will correctly take their own hat!

Quicksort: An Introduction

Goal: Another sorting algorithm that uses divide-and-conquer

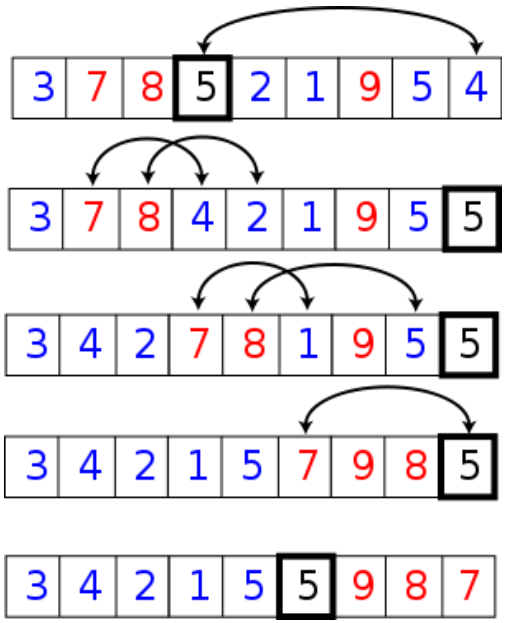


- Idea:
  - Select an element in the array
  - Partition the other elements of the array around it
- Is the array more sorted than it was before?
- Answer: yes!
- Next step: recursively sort the left and right sides of the array as well.

Problem: What about "adversarial inputs"? This algorithm will perform better on some inputs than others.

Quicksort: Randomized

Can we write an algorithm for sorting that uses coin tossing (randomness)?



- Idea:
  - *Randomly* select an element in the array
  - Partition the other elements of the array around it
  - Recursively sort the left and right sides of the array

Result: Another divide and conquer algorithm for sorting, that uses **randomness.**

Video 1.5
Sampath Kannan

Quicksort

Idea: Choose an element at random. Partition the array around this element. Recursively sort the left and right side.

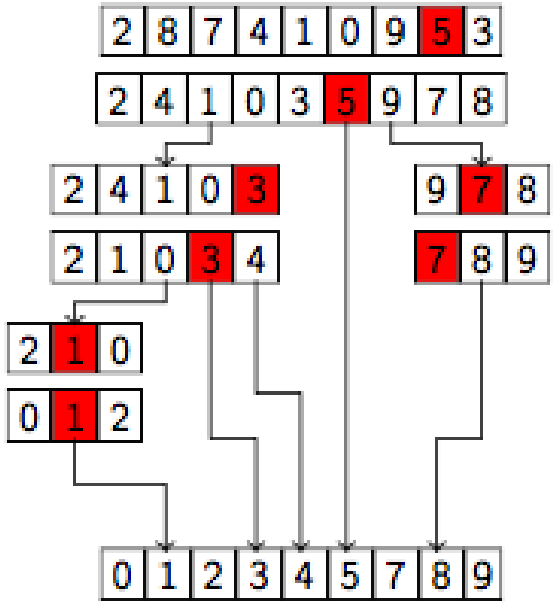
Quicksort

Idea: Choose an element at random. Partition the array around this element. Recursively sort the left and right side.

```
quicksort(A):
    quicksort(A, 0, len(A)-1)

quicksort(A, lo, hi):
    if (hi >= lo) return
    pivot_location <- partition(A, lo, hi)
    quicksort(A, lo, pivot_location - 1)
    quicksort(A, pivot_location + 1, hi)

partition(A, lo, hi):
    pivot_index <- random(lo, hi)
    swap(A, pivot_index, hi)
    pivot <- A[hi]
    i <- lo, j <- hi, C <- new array
    for k = lo to hi-1:
        if A[k] <= pivot:
            C[i++] <- A[k]
        else:
            C[j--] <- A[k]
        k++
    C[i] <- A[hi] (copy the pivot in)
    copy C[lo:hi] back into A
    return i
```



Quicksort

Quicksort (compare to Mergesort)

Quicksort

Quicksort (compare to Mergesort)

- divide-and-conquer algorithm
- First partition, then sort recursively

Quicksort

Quicksort (compare to Mergesort)

- divide-and-conquer algorithm
- First partition, then sort recursively
- Can be done with no extra space
- runtime: See next slide

Quicksort: Analysis

- First: the recurrence for quicksort
- Step 1: Partition requires $O(n)$
- Step 2: Recursively sort left and right sides of the array

Quicksort: Analysis

- First: the recurrence for quicksort
- Step 1: Partition requires $O(n)$
- Step 2: Recursively sort left and right sides of the array
  - What are the sizes of these two arrays?
  - k and $n - k - 1$, for some k
- $T(n) = T(k) + T(n - k - 1) + O(n)$

Quicksort: Analysis

- First: the recurrence for quicksort
- Step 1: Partition requires $O(n)$
- Step 2: Recursively sort left and right sides of the array
  - What are the sizes of these two arrays?
  - k and $n - k - 1$, for some k
- $T(n) = T(k) + T(n - k - 1) + O(n)$

Worst case (bad partition):

- partition does not split array at all
  at every step ($k = 1$ or $n - 1$)
- $T(n) = T(1) + T(n - 1) + n$
- $T(n) = O(n^2)$ (similar to insertion sort)

Quicksort: Analysis

- First: the recurrence for quicksort
- Step 1: Partition requires $O(n)$
- Step 2: Recursively sort left and right sides of the array
  - What are the sizes of these two arrays?
  - k and $n - k - 1$, for some k
- $T(n) = T(k) + T(n - k - 1) + O(n)$

Worst case (bad partition):

- partition does not split array at all at every step ($k = 1$ or $n - 1$)
- $T(n) = T(1) + T(n - 1) + n$
- $T(n) = O(n^2)$ (similar to insertion sort)

Best case (good partition):

- partition splits array evenly at ever step ($k = \frac{n}{2}$)
- $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(n)$
- $T(n) = O(n \lg n)$ (recall from mergesort)

Quicksort: Analysis

- First: the recurrence for quicksort
- Step 1: Partition requires $O(n)$
- Step 2: Recursively sort left and right sides of the array
  - What are the sizes of these two arrays?
  - k and $n - k - 1$, for some k
- $T(n) = T(k) + T(n - k - 1) + O(n)$

Worst case (bad partition):

- partition does not split array at all at every step ($k = 1$ or $n - 1$)
- $T(n) = T(1) + T(n - 1) + n$
- $T(n) = O(n^2)$ (similar to insertion sort)

Best case (good partition):

- partition splits array evenly at ever step ($k = \frac{n}{2}$)
- $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(n)$
- $T(n) = O(n \lg n)$ (recall from mergesort)

How does the algorithm perform on average?
We can analyze with **expectation**

Quicksort: Analysis

Recurrence for quicksort:

Quicksort: Analysis

Recurrence for quicksort:

- taking the expected value over all possible i:

Quicksort: Analysis

Recurrence for quicksort:

- taking the expected value over all possible i:
- $T(n) = \frac{1}{n} \sum_{i=1}^{n} T(i) + T(n-i) + O(n)$

Quicksort: Analysis

Recurrence for quicksort:

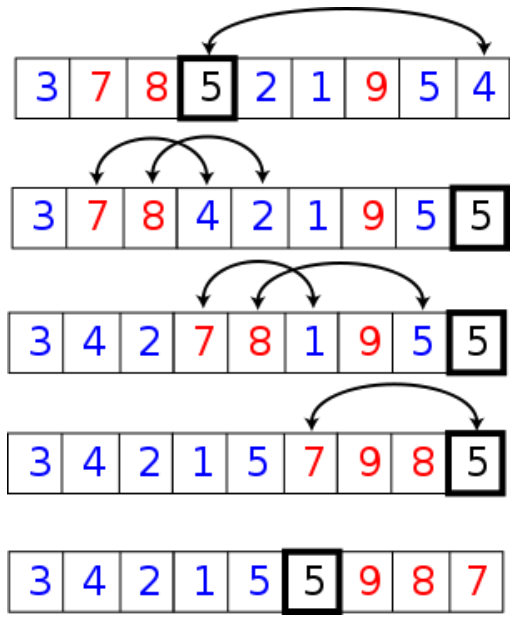
- taking the expected value over all possible i:
- $T(n) = \frac{1}{n} \sum_{i=1}^{n} T(i) + T(n-i) + O(n)$
- This is difficult to analyze! Can we find a better way to analyze quicksort?

Quicksort: Analysis

Recurrence for quicksort:

- taking the expected value over all possible i:
- $T(n) = \frac{1}{n} \sum_{i=1}^{n} T(i) + T(n-i) + O(n)$
- This is difficult to analyze! Can we find a better way to analyze quicksort?

Partition step



Idea: Any two elements are never compared more than once

- What happens after an element is compared to the partitioning element?
  - these two elements won't be compared again

Quicksort: Analysis

Analyze with random variables:

- denote the kth smallest element in the array as e_k

Quicksort: Analysis

Analyze with random variables:

- denote the kth smallest element in the array as e_k
- X = total number of comparisons
-

$$X_{ij} = \begin{cases} 1 & e_i \text{ and } e_j \text{ are compared} \\ 0 & e_i \text{ and } e_j \text{ are not compared} \end{cases}$$

Quicksort: Analysis

Analyze with random variables:

- denote the kth smallest element in the array as e_k
- X = total number of comparisons
-

$$X_{ij} = \begin{cases} 1 & e_i \text{ and } e_j \text{ are compared} \\ 0 & e_i \text{ and } e_j \text{ are not compared} \end{cases}$$

- Then $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

Quicksort: Analysis

Analyze with random variables:

- denote the kth smallest element in the array as e_k
- X = total number of comparisons
- $$X_{ij} = \begin{cases} 1 & e_i \text{ and } e_j \text{ are compared} \\ 0 & e_i \text{ and } e_j \text{ are not compared} \end{cases}$$
- Then $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
- $E[X] = E[\Sigma \Sigma X_{ij}] = \Sigma \Sigma E[X_{ij}]$ by lin. of exp.
- Recall: $E[X_{ij}] = 1 * P(X_{ij} = 1) + 0 * P(X_{ij} = 0)$
- $E[X_{ij}] = P(X_{ij} = 1)$

Quicksort: Analysis

Analyze with random variables:

What is the probability that e_i and e_j are compared?

Quicksort: Analysis

Analyze with random variables:

What is the probability that e_i and e_j are compared?

- e_i and e_j will be compared if either is selected as a pivot

Quicksort: Analysis

Analyze with random variables:

What is the probability that e_i and e_j are compared?

- e_i and e_j will be compared if either is selected as a pivot
- e_i and e_j will not be compared if some e_k, $i < k < j$ is selected as a pivot **first**
  - e_i will be to the left of e_k, and e_j will be to the right.

Quicksort: Analysis

- Which pivots must be chosen for e_i and e_j to be compared?
  - either e_i or e_j (2 total)

Quicksort: Analysis

- Which pivots must be chosen for e_i and e_j to be compared?
  - either e_i or e_j (2 total)
- Which pivots for e_i and e_j not to be compared?
  - $e_{i+1}, e_{i+2}, \ldots,$ or e_{j-1} ($j - i - 1$ total)

Quicksort: Analysis

- Which pivots must be chosen for e_i and e_j to be compared?
  - either e_i or e_j (2 total)
- Which pivots for e_i and e_j not to be compared?
  - $e_{i+1}, e_{i+2}, \ldots,$ or e_{j-1} ($j - i - 1$ total)
- Elements are chosen as pivots randomly
- $E[X_{ij}] = \frac{2}{(j-i-1)+2} = \frac{2}{j-i+1}$
- $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$
- $E[X] \leq 2nlgn \in O(nlgn)$

Quicksort: Analysis

- Which pivots must be chosen for e_i and e_j to be compared?
  - either e_i or e_j (2 total)
- Which pivots for e_i and e_j not to be compared?
  - $e_{i+1}, e_{i+2}, \ldots,$ or e_{j-1} ($j - i - 1$ total)
- Elements are chosen as pivots randomly
- $E[X_{ij}] = \frac{2}{(j-i-1)+2} = \frac{2}{j-i+1}$
- $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$
- $E[X] \leq 2nlgn \in O(nlgn)$

Result: Randomized Quicksort makes an expected $O(nlgn)$ comparisons!

Quick Select

Goal: select the kth smallest element of an array

Quick Select

Goal: select the kth smallest element of an array

Option 1:

- Use quicksort to sort the array A
- Select the kth smallest element ($A[k-1]$)
- Time required: $O(nlgn)$ to sort the array
- Are we doing unnecessary work? Can we do better?

Quick Select

Goal: select the kth smallest element of an array

Option 1:

- Use quicksort to sort the array A
- Select the kth smallest element ($A[k-1]$)
- Time required: $O(nlgn)$ to sort the array
- Are we doing unnecessary work? Can we do better?

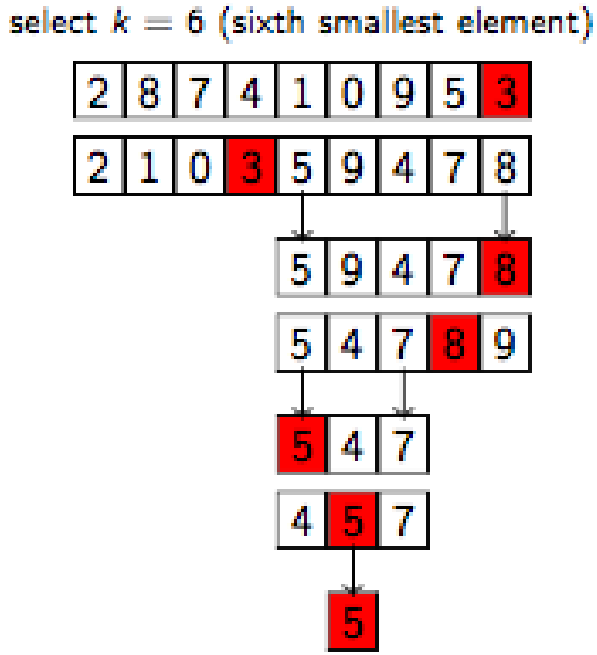
Key Idea:

- When we partition the array, the kth smallest element will only be on one side of this partition
- No need to recursively sort both sides of the array: Only the side containing the element we want

Quickselect

```
quickselect(A, k):
    quickselect(A, 0, len(A)-1, k)

quickselect(A, lo, hi, k):
    if (hi == lo) return A[lo]
    pivot_location <- partition(A, lo, hi)
    if pivot_location == k:
        return A[k]
    else if pivot_location < k:
        return quickselect(A, lo, pivot_location - 1, k)
    else :
        quickselect(A, pivot_location + 1, hi, k - )
```

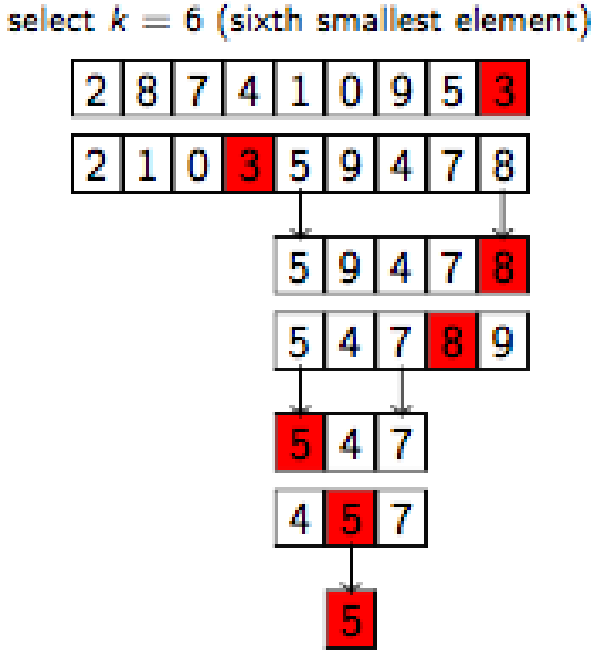


Quickselect

```
quickselect(A, k):
    quickselect(A, 0, len(A)-1, k)

quickselect(A, lo, hi, k):
    if (hi == lo) return A[lo]
    pivot_location <- partition(A, lo, hi)
    if pivot_location == k:
        return A[k]
    else if pivot_location < k:
        return quickselect(A, lo, pivot_location - 1, k)
    else :
        quickselect(A, pivot_location + 1, hi, k - )
```

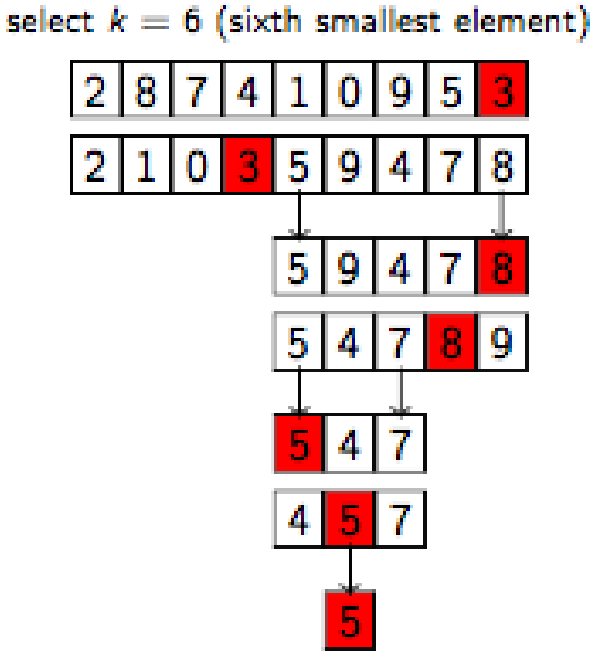
- Analysis?



Quickselect

```
quickselect(A, k):
    quickselect(A, 0, len(A)-1, k)

quickselect(A, lo, hi, k):
    if (hi == lo) return A[lo]
    pivot_location <- partition(A, lo, hi)
    if pivot_location == k:
        return A[k]
    else if pivot_location < k:
        return quickselect(A, lo, pivot_location - 1, k)
    else :
        quickselect(A, pivot_location + 1, hi, k - )
```



- Analysis?
  - We will use a similar analysis to Quicksort
  - What will change? Are certain elements less likely to be compared?

Quickselect: Analysis

Analyze with random variables:

Quickselect: Analysis

Analyze with random variables:

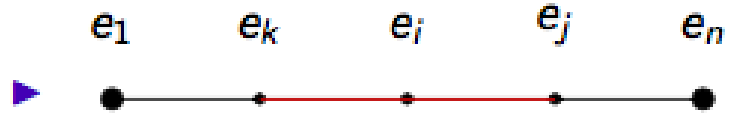
- denote the kth smallest element in the array as e_k
- What is the probability that e_i and e_j are compared *when selecting* e_k ?
- 3 cases:

Quickselect: Analysis

Analyze with random variables:

- denote the kth smallest element in the array as e_k
- What is the probability that e_i and e_j are compared *when selecting* e_k ?
- 3 cases:

- case 1: $k < i < j$.
  - *Compared when*: e_i or e_j are selected as pivots
  - *Not compared when*: any other element between e_k and e_j are selected
  - $P(e_i, e_j \text{ compared}) = \frac{2}{j-k+1}$

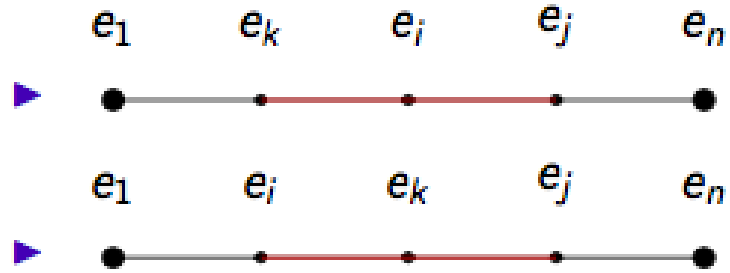


Quickselect: Analysis

Analyze with random variables:

- denote the kth smallest element in the array as e_k
- What is the probability that e_i and e_j are compared *when selecting* e_k ?
- 3 cases:

- case 1: $k < i < j$.
  - *Compared when*: e_i or e_j are selected as pivots
  - *Not compared when*: any other element between e_k and e_j are selected
  - $P(e_i \ e_j \ compared) = \frac{2}{j-k+1}$
- case 2: $i < k < j$.
  - Similarly: $P(e_i \ e_j \ compared) = \frac{2}{j-i+1}$

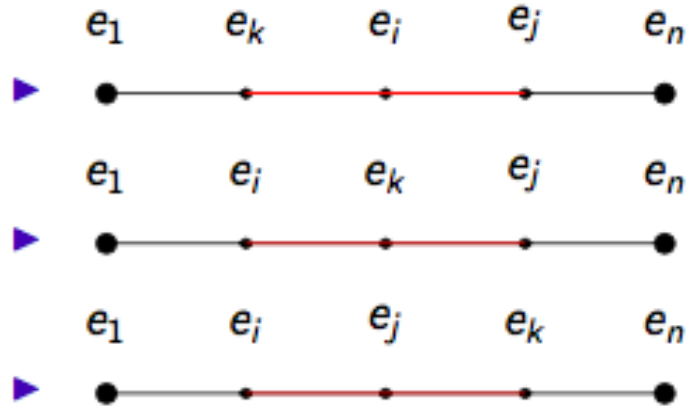


Quickselect: Analysis

Analyze with random variables:

- denote the kth smallest element in the array as e_k
- What is the probability that e_i and e_j are compared *when selecting* e_k ?
- 3 cases:

- case 1: $k < i < j$.
  - *Compared when*: e_i or e_j are selected as pivots
  - *Not compared when*: any other element between e_k and e_j are selected
  - $P(e_i \ e_j \ \textit{compared}) = \frac{2}{j-k+1}$
- case 2: $i < k < j$.
  - Similarly: $P(e_i \ e_j \ \textit{compared}) = \frac{2}{j-i+1}$
- case 3: $i < j < k$.
  - Similarly: $P(e_i \ e_j \ \textit{compared}) = \frac{2}{k-i+1}$



Quickselect: Analysis

Runtime:

- Similar to quicksort analysis, how many total comparisons are we making?

Quickselect: Analysis

Runtime:

- Similar to quicksort analysis, how many total comparisons are we making?
- Sum over all pairs of elements e_i, e_j (split among the 3 cases)

$$E[X] = \sum_{i<j<k} \frac{2}{k-i+1} + \sum_{i<k<j} \frac{2}{j-i+1} + \sum_{k<i<j} \frac{2}{j-k+1}$$

- Non obvious sum, but yields $E[X] = O(n)$!

Quickselect: Analysis

Runtime:

- Similar to quicksort analysis, how many total comparisons are we making?
- Sum over all pairs of elements e_i, e_j (split among the 3 cases)

$$E[X] = \sum_{i<j<k} \frac{2}{k-i+1} + \sum_{i<k<j} \frac{2}{j-i+1} + \sum_{k<i<j} \frac{2}{j-k+1}$$

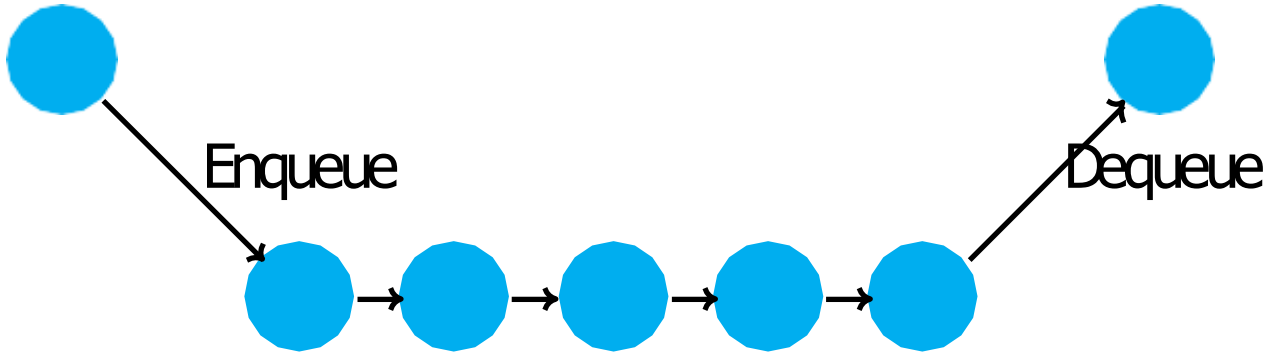
- Non obvious sum, but yields $E[X] = O(n)$!

Outcome:

- Quickselect is faster than quicksort!
- Note: quickselect is randomized
- Can we make it deterministic, and still keep the worst case $O(n)$?
- Yes, with some extra work

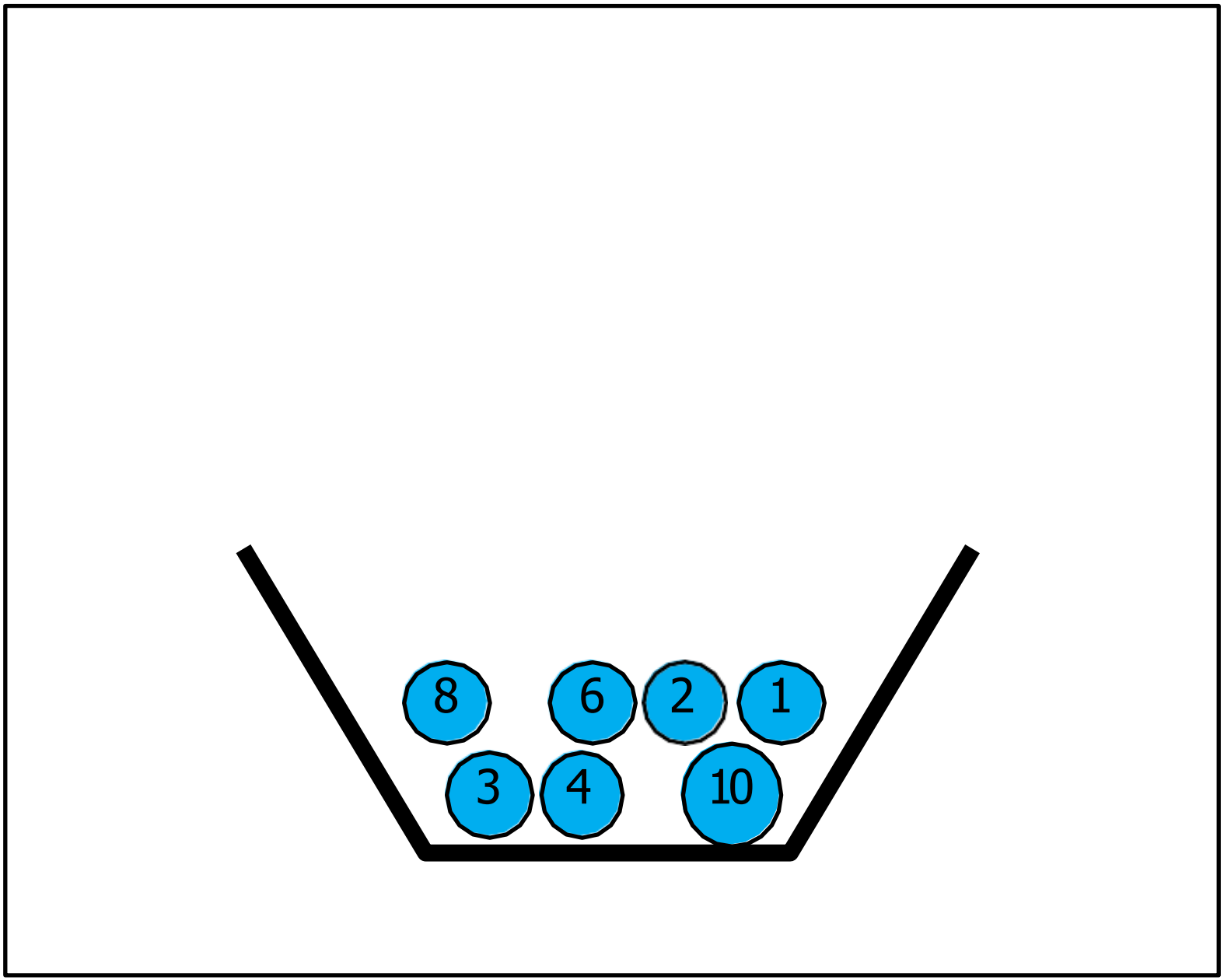
Video 1.6
Sampath Kannan

Queues

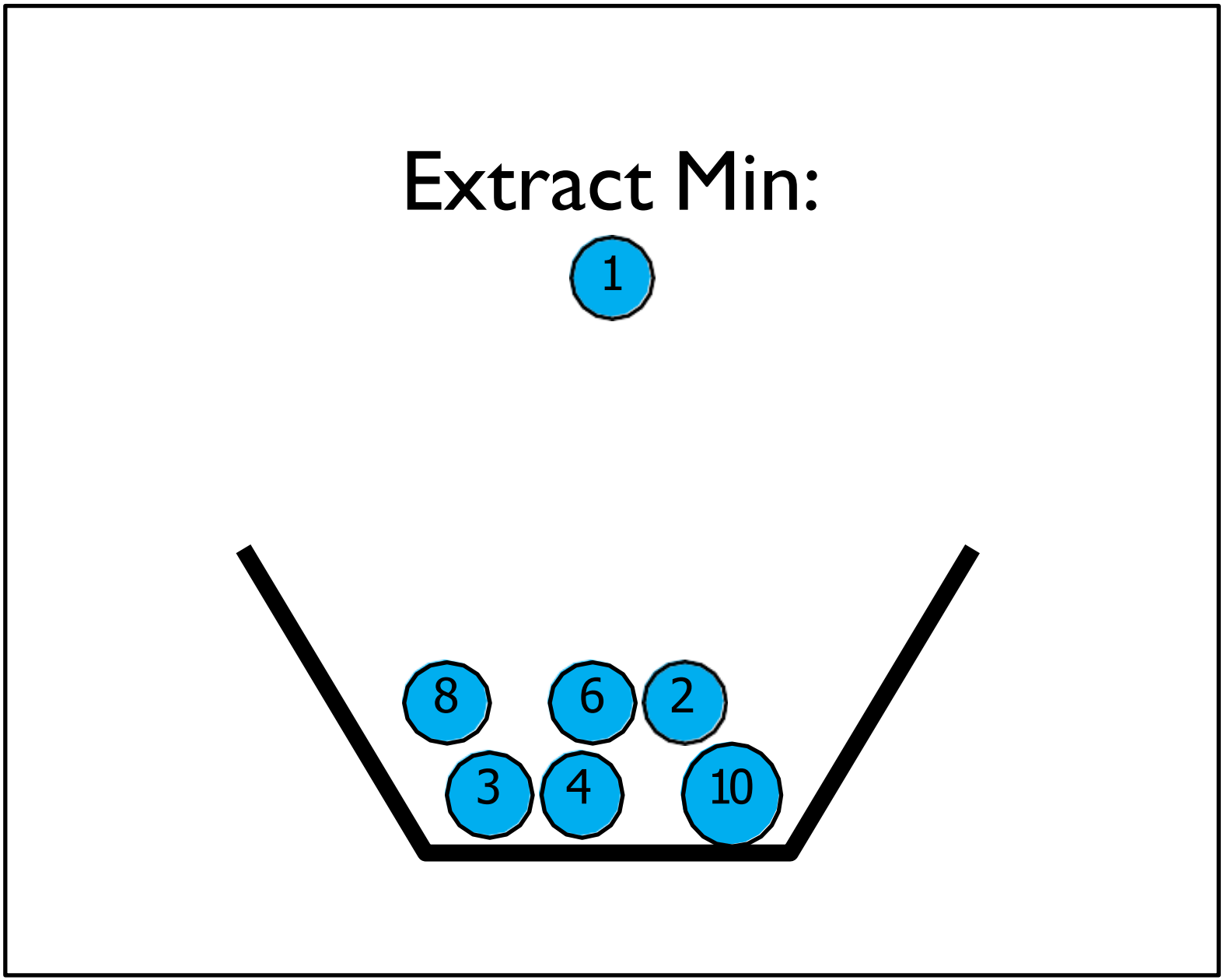


- Sometimes we want to extract elements not in the order we insert them but instead in the order of some given keys. We call this a *priority queue*
- For example your operating system is constantly getting jobs to complete, it needs a fast way of getting the highest priority job to schedule next

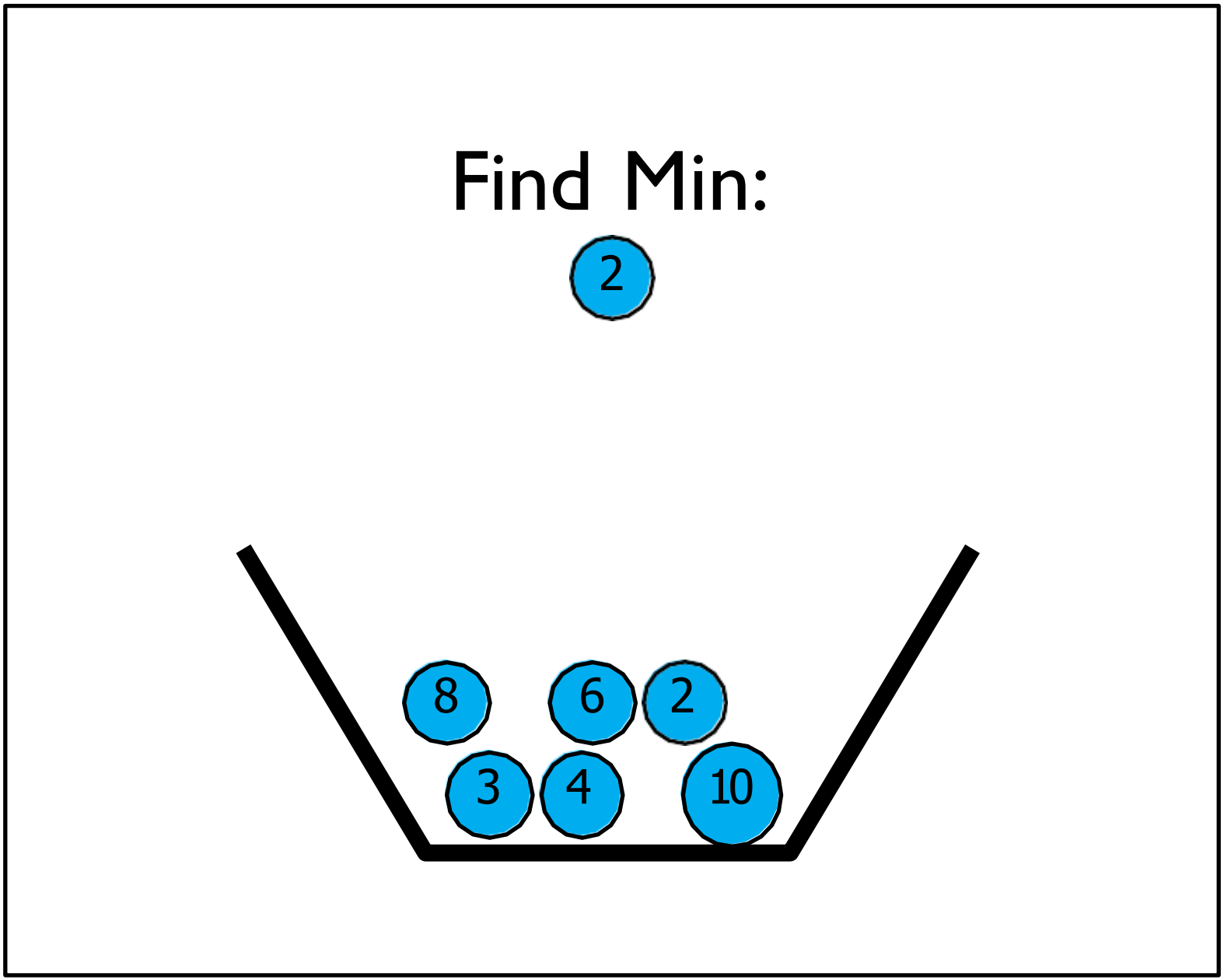
Operations of Priority Queues



Operations of Priority Queues

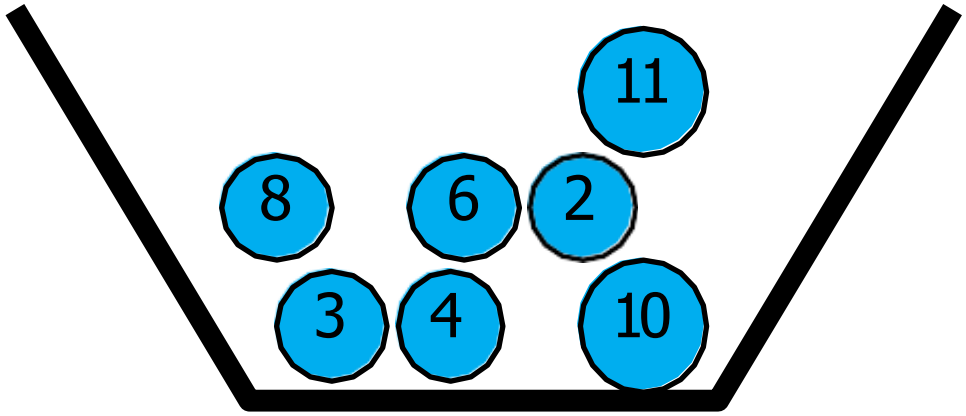


Operations of Priority Queues



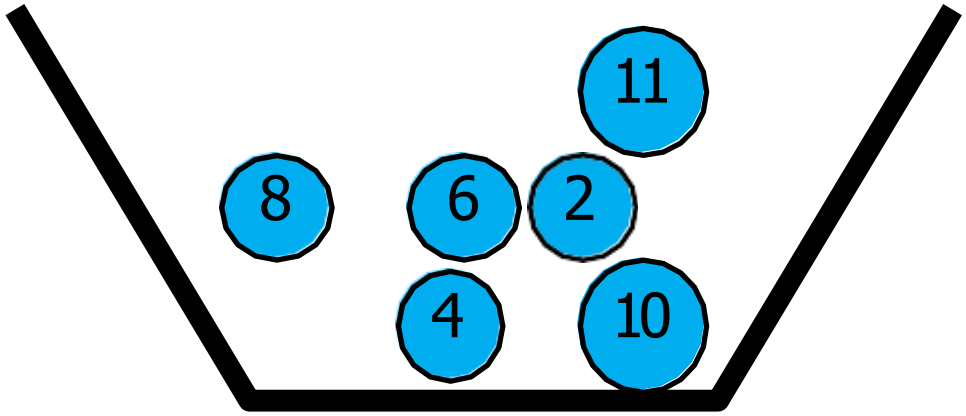
Operations of Priority Queues

Insert(11):



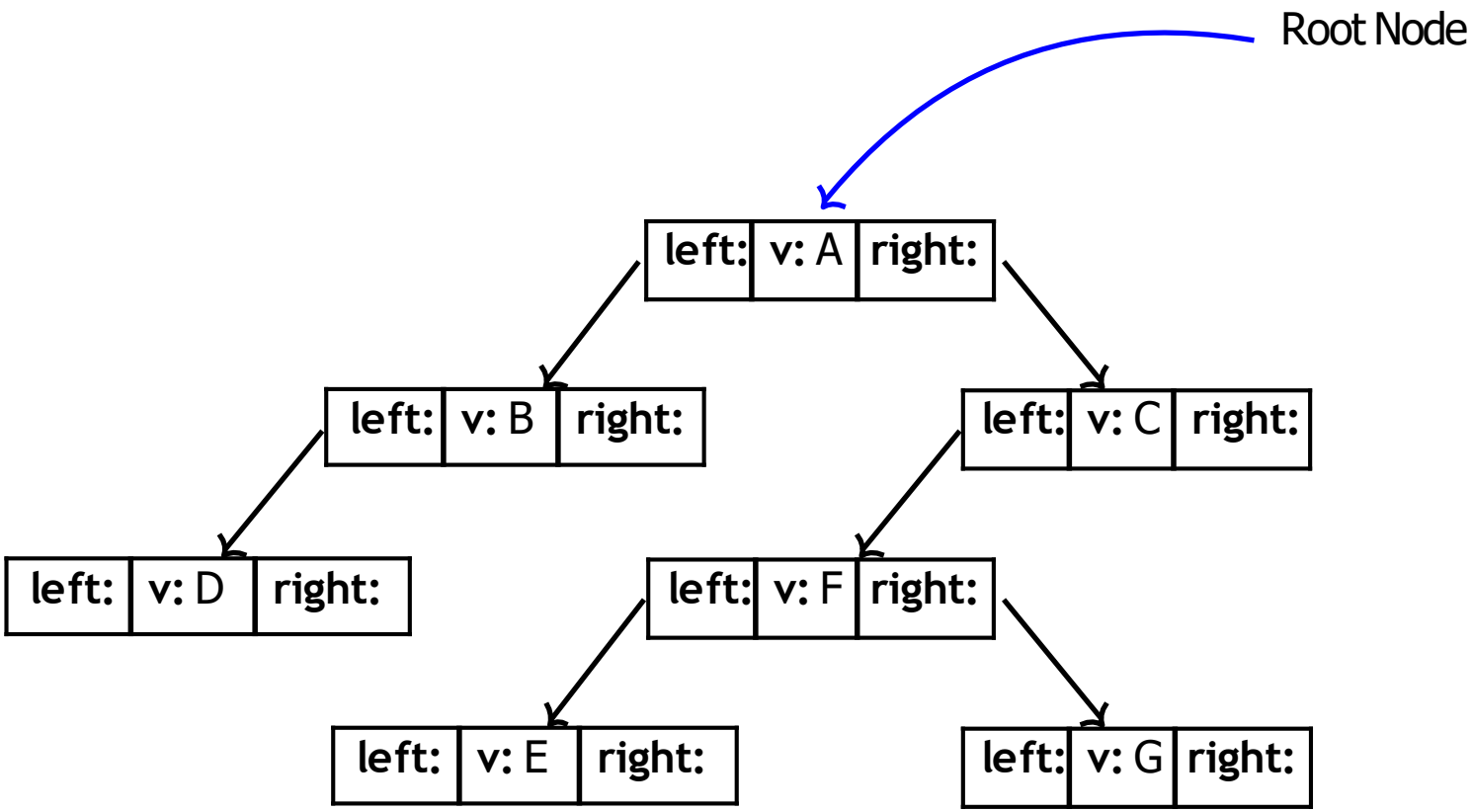
Operations of Priority Queues

Delete(3):



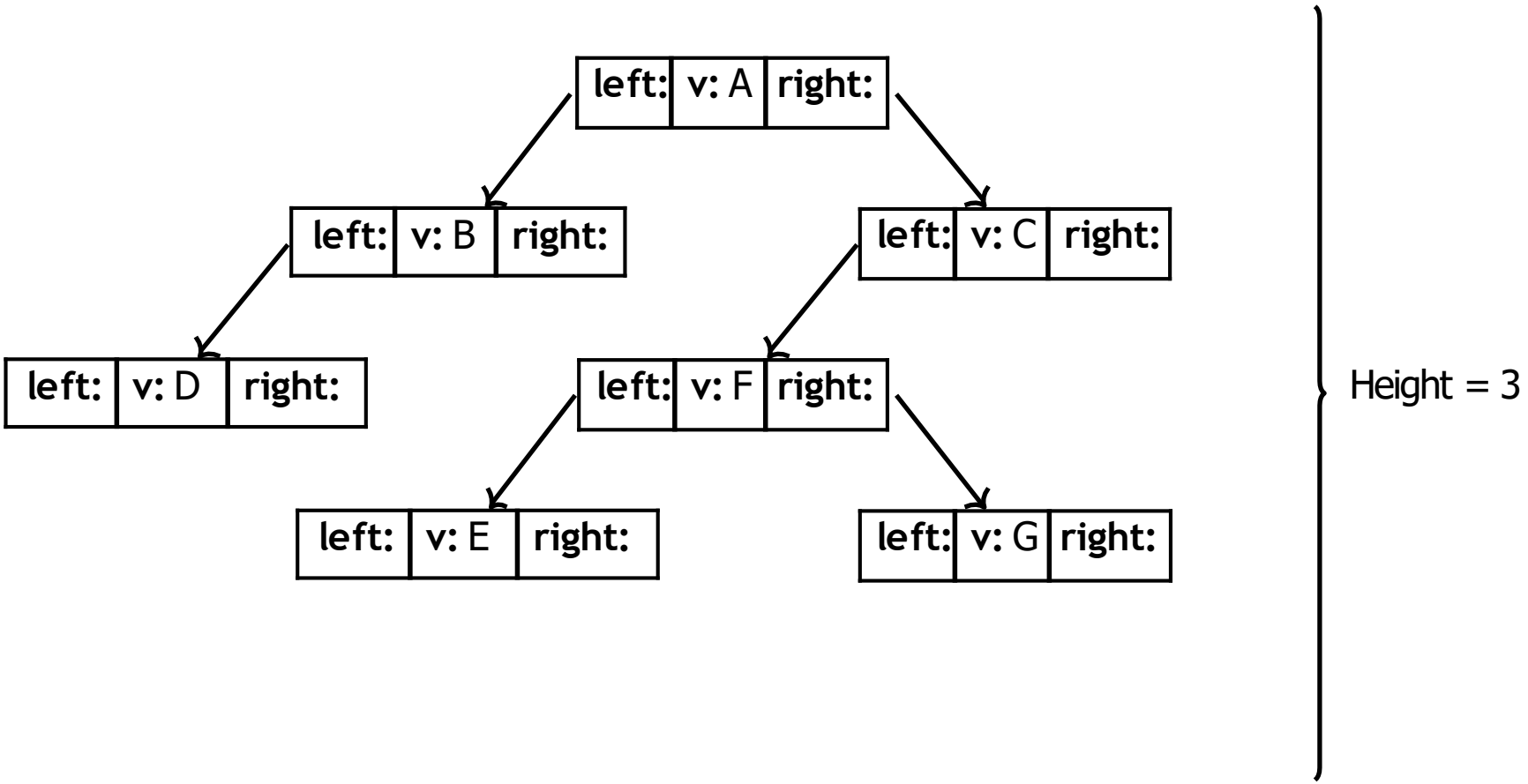
Trees

In order to make an efficient priority heap we will use a more general data structure called a tree.



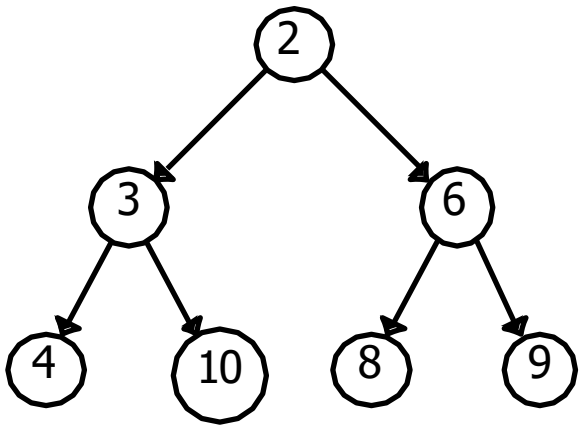
Trees

In order to make an efficient priority heap we will use a more general data structure called a tree.



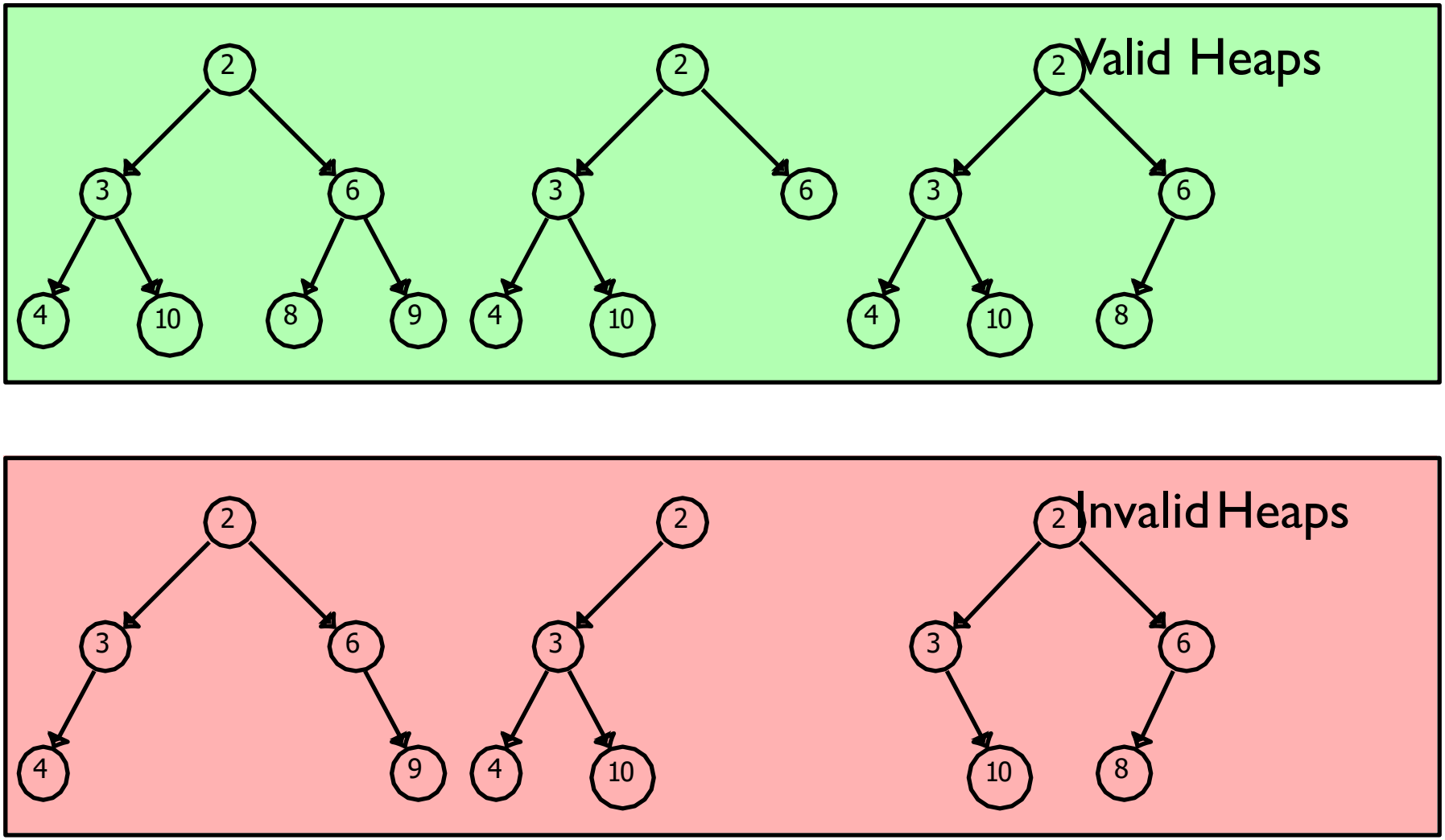
Heaps as trees

We can use a tree to make a heap by enforcing the properties that node will have a key value that is less than both of it's children, and that the tree will always be complete except for the last layer.



- This makes finding the minimum very easy. It's always on top!
- We will see that removing the root (minimum) element can be done in a number of operations proportional to the height.
- However if we want to find an arbitrary element we will have to search the whole tree.

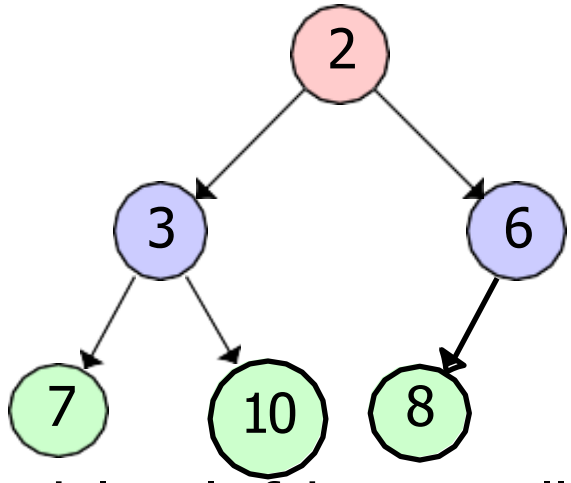
Heaps Shapes



Video 1.7
Sampath Kannan

Heap Representation

Since the tree for a heap will always be contiguous we can represent them implicitly with an array



2	3	6	7	10	8

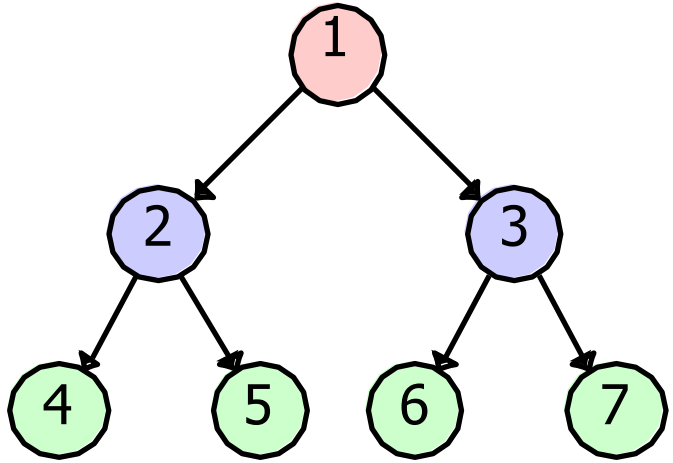
So the i th level of the tree will occupy spots 2^{i-1} to $2^i - 1$ (we are using 1 based indexing for convenience)

Heap Representation

We need to be able to compute positions of the left and right children of a given element.

Heap Representation

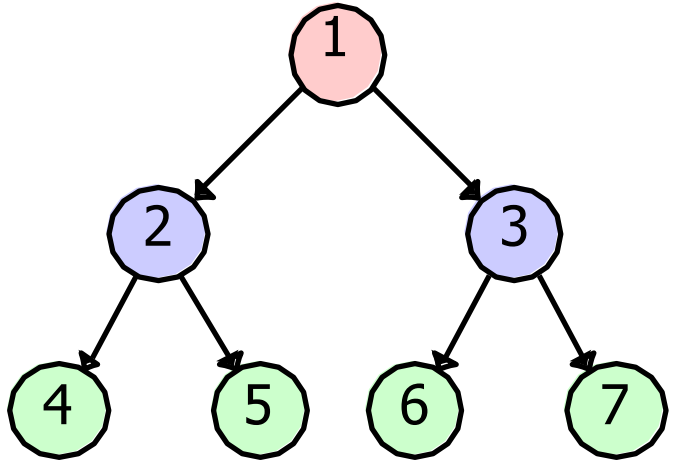
We need to be able to compute positions of the left and right children of a given element.



- Left child of 1 is 2, left child of 2 is 4, left child of 3 is 6, etc...

Heap Representation

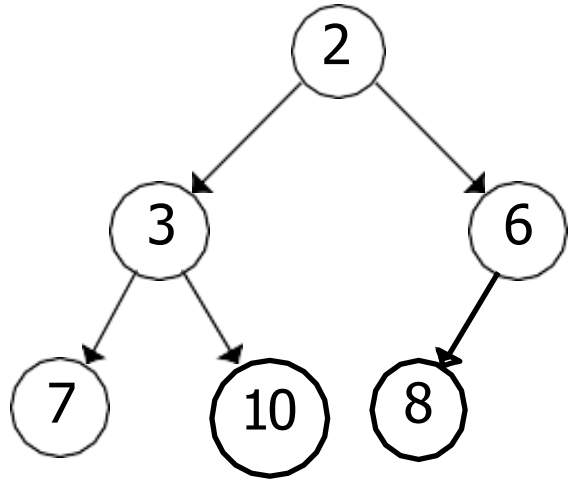
We need to be able to compute positions of the left and right children of a given element.



- Left child of 1 is 2, left child of 2 is 4, left child of 3 is 6, etc...
- In general the left child of node k is at position $2k$. So the right child is at $2k + 1$

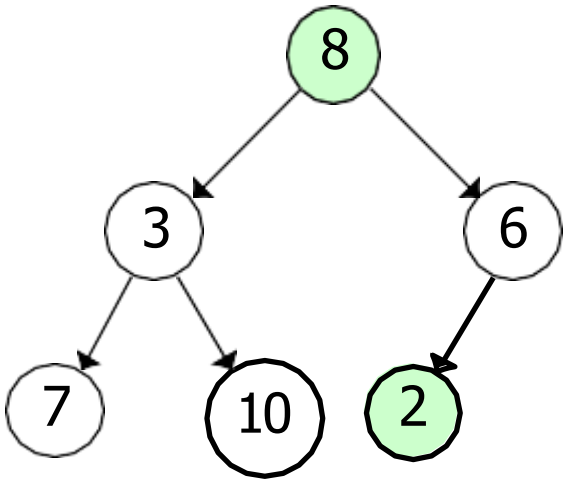
Operations on Heaps: Extract Min

We want to remove the minimum element (root) while maintaining the two heap properties: order and shape



Operations on Heaps: Extract Min

Step 1: Swap the root node with the node in the bottom right

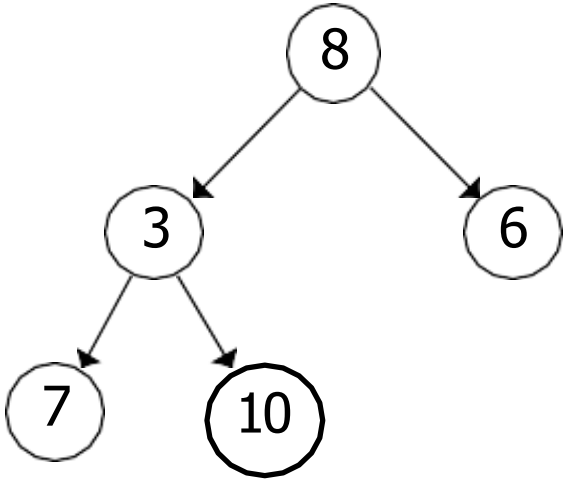



```
sink(A, k):
  N = length(A)
  while 2*k <= N
    smallest = 2*k
    if A[2*k] < A[2*k+1]
       smallest = 2*k+1
    if A[k] < smallest: break
    swap(A[k], A[smallest])
    k = smallest

extract-min(A, k):
  N = length(A)
  min = A[1]
  A[1] = A[N]
  sink(A, 1)
  return min
```

Operations on Heaps: Extract Min

Step 2: Now we can remove (2) while maintaining the shape property

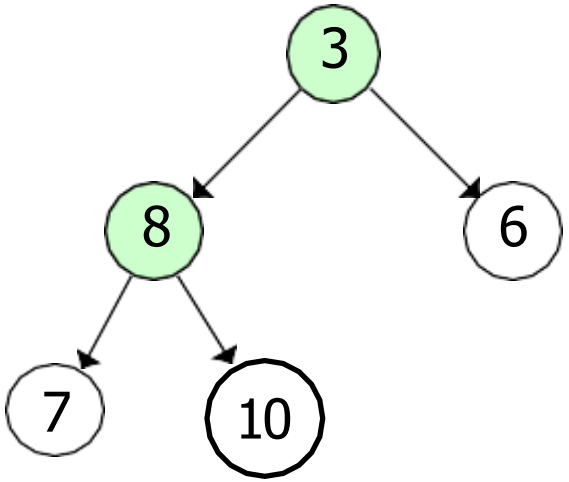



```
sink(A, k):
  N = length(A)
  while 2*k <= N
    smallest = 2*k
    if A[2*k] < A[2*k+1]
       smallest = 2*k+1
    if A[k] < smallest: break
    swap(A[k], A[smallest])
    k = smallest

extract-min(A, k):
  N = length(A)
  min = A[1]
  A[1] = A[N]
  sink(A, 1)
  return min
```

Operations on Heaps: Extract Min

Step 3: We will fix the order property by swapping (8) with it's smallest child

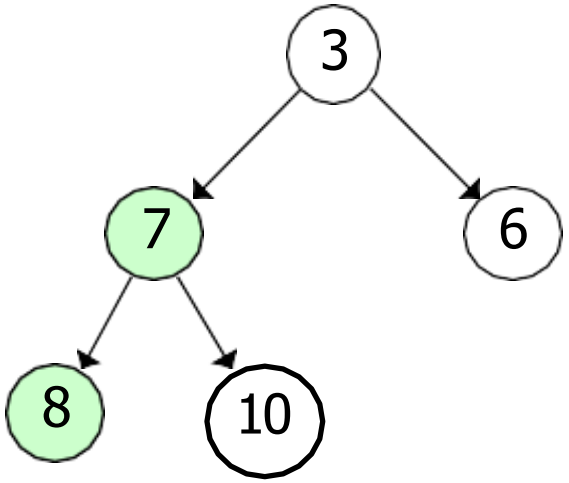



```
sink(A, k):
  N = length(A)
  while 2*k <= N
    smallest = 2*k
    if A[2*k] < A[2*k+1]
       smallest = 2*k+1
    if A[k] < smallest: break
    swap(A[k], A[smallest])
    k = smallest

extract-min(A, k):
  N = length(A)
  min = A[1]
  A[1] = A[N]
  sink(A, 1)
  return min
```

Operations on Heaps: Extract Min

Step 4: Keep fixing the order property by swapping (8) with it's smallest child again

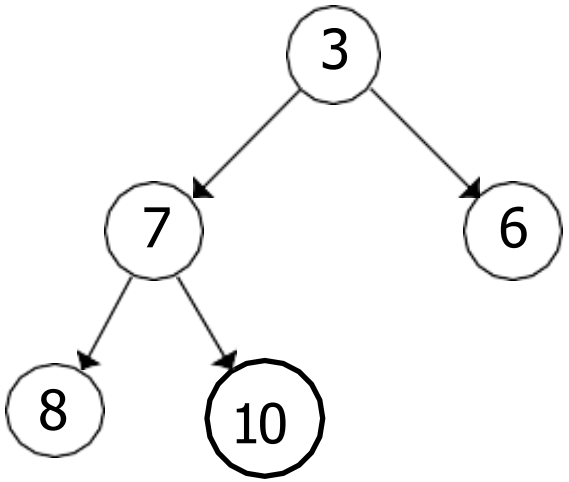



```
sink(A, k):
  N = length(A)
  while 2*k <= N
    smallest = 2*k
    if A[2*k] < A[2*k+1]
       smallest = 2*k+1
    if A[k] < smallest: break
    swap(A[k], A[smallest])
    k = smallest

extract-min(A, k):
  N = length(A)
  min = A[1]
  A[1] = A[N]
  sink(A, 1)
  return min
```

Operations on Heaps: Extract Min

Step 5: The heap properties have been preserved so we're done!

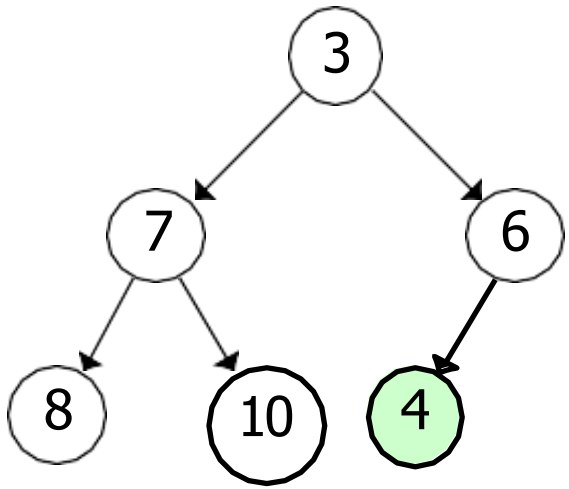



```
sink(A, k):
  N = length(A)
  while 2*k <= N
    smallest = 2*k
    if A[2*k] < A[2*k+1]
       smallest = 2*k+1
    if A[k] < smallest: break
    swap(A[k], A[smallest])
    k = smallest

extract-min(A, k):
  N = length(A)
  min = A[1]
  A[1] = A[N]
  sink(A, 1)
  return min
```

Operations on Heaps: Insert

Step 1: Preserve the shape property by inserting the new element at the bottom right

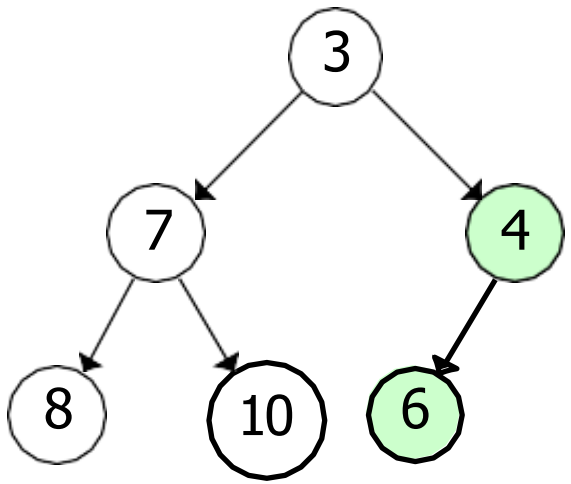



```
swim(A, k):
    while k > 1 and A[k/2] < A[k]:
        swap(A[k], A[k/2])
        k = k/2

insert(A, k, val):
    N = length(A)
    A[N+1] = val
    swim(A, N+1)
```

Operations on Heaps: Insert

Step 2: Fix the order property by swapping (4) with its parent since it's smaller

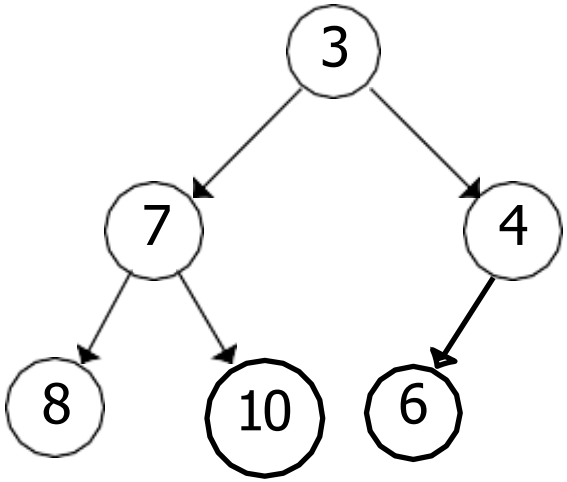



```
swim(A, k):
    while k > 1 and A[k/2] < A[k]:
        swap(A[k], A[k/2])
        k = k/2

insert(A, k, val):
    N = length(A)
    A[N+1] = val
    swim(A, N+1)
```

Operations on Heaps: Insert

Step 3: (4) is bigger than its parent now so we're done!




```
swim(A, k):
    while k > 1 and A[k/2] < A[k]:
        swap(A[k], A[k/2])
        k = k/2

insert(A, k, val):
    N = length(A)
    A[N+1] = val
    swim(A, N+1)
```

Heap efficiency

- All operations on the heap are a combination of a constant number of operations and sink or swim operation.
- Swim operation executes as long as $k > 1$ and divides it by 2 on every iteration
- Can execute at most $\log_2 k$ times. Since k is initially at most n, the number of elements, swim has a run time that is $O(\log n)$
- By the same logic sink has run time that is $O(\log n)$ as well.
- So all the operations are $O(\log n)$. Except for delete which must first take potentially $O(n)$ steps to locate the given element in the array.

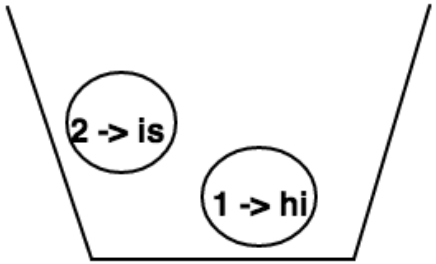
Video 1.8
Sampath Kannan

Dynamic Dictionaries

Dynamic Dictionaries support three main operations:

- **insert** into a dictionary
- **delete** from a dictionary
- **search** for an element in a dictionary

Abstract representation:



Dynamic dictionaries are used in applications everywhere:

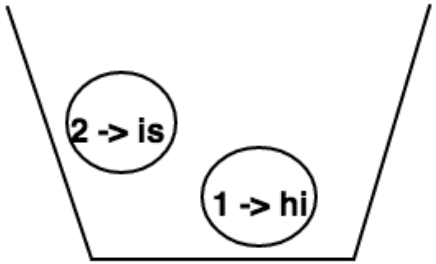
- Databases
- Router lookup tables, ids of IP packets
- Any application that involves storing information!

Dynamic Dictionaries

Dynamic Dictionaries support three main operations:

- **insert** into a dictionary
- **delete** from a dictionary
- **search** for an element in a dictionary

Abstract representation:



next: insert the pair (3, "the")

Dynamic dictionaries are used in applications everywhere:

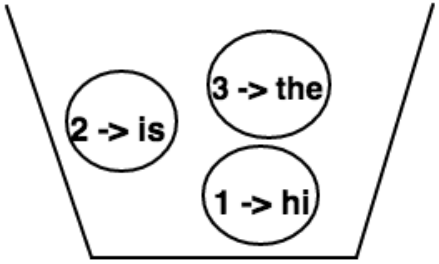
- Databases
- Router lookup tables, ids of IP packets
- Any application that involves storing information!

Dynamic Dictionaries

Dynamic Dictionaries support three main operations:

- **insert** into a dictionary
- **delete** from a dictionary
- **search** for an element in a dictionary

Abstract representation:



Dynamic dictionaries are used in applications everywhere:

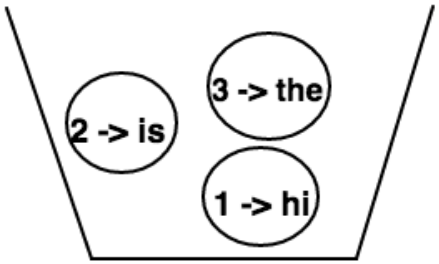
- Databases
- Router lookup tables, ids of IP packets
- Any application that involves storing information!

Dynamic Dictionaries

Dynamic Dictionaries support three main operations:

- **insert** into a dictionary
- **delete** from a dictionary
- **search** for an element in a dictionary

Abstract representation:



next: lookup 1

Dynamic dictionaries are used in applications everywhere:

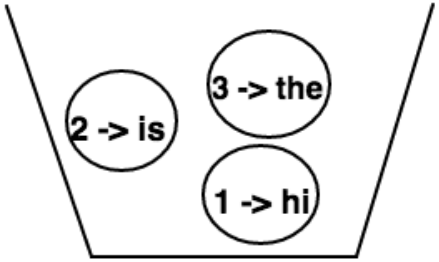
- Databases
- Router lookup tables, ids of IP packets
- Any application that involves storing information!

Dynamic Dictionaries

Dynamic Dictionaries support three main operations:

- **insert** into a dictionary
- **delete** from a dictionary
- **search** for an element in a dictionary

Abstract representation:



next: lookup 1 returns "hi"

Dynamic dictionaries are used in applications everywhere:

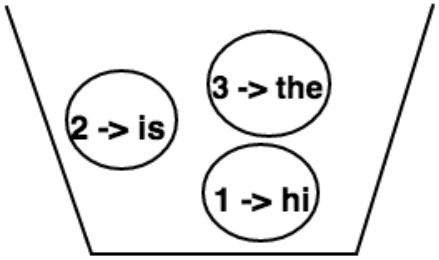
- Databases
- Router lookup tables, ids of IP packets
- Any application that involves storing information!

Dynamic Dictionaries

Dynamic Dictionaries support three main operations:

- **insert** into a dictionary
- **delete** from a dictionary
- **search** for an element in a dictionary

Abstract representation:



next: lookup 1
returns "hi"
next: delete 3 from dictionary

Dynamic dictionaries are used in applications everywhere:

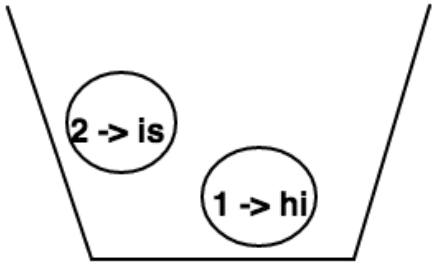
- Databases
- Router lookup tables, ids of IP packets
- Any application that involves storing information!

Dynamic Dictionaries

Dynamic Dictionaries support three main operations:

- **insert** into a dictionary
- **delete** from a dictionary
- **search** for an element in a dictionary

Abstract representation:



Dynamic dictionaries are used in applications everywhere:

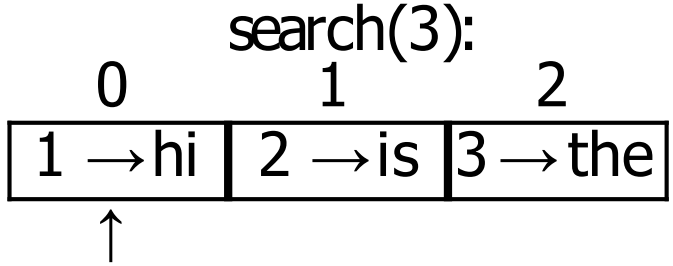
- Databases
- Router lookup tables, ids of IP packets
- Any application that involves storing information!

Implementations of Dictionaries

Can we find an efficient implementation for dictionaries?

Attempt 1: Arrays

- search: $O(n)$
  - entire array must be traversed
- insertion, deletion: $O(n)$
  - Array may need to be resized (requires copying all elements to a new array

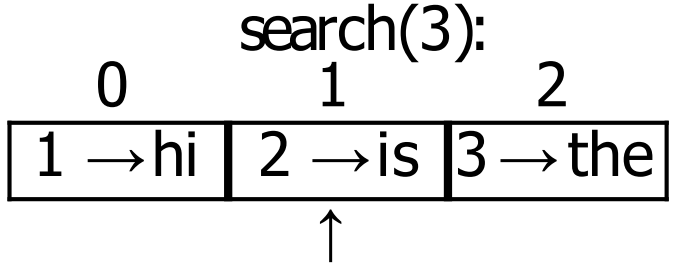


Implementations of Dictionaries

Can we find an efficient implementation for dictionaries?

Attempt 1: Arrays

- search: $O(n)$
  - entire array must be traversed
- insertion, deletion: $O(n)$
  - Array may need to be resized (requires copying all elements to a new array

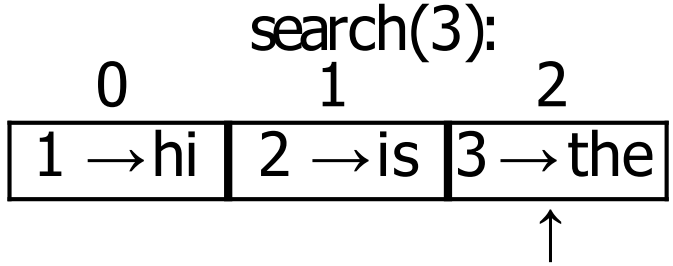


Implementations of Dictionaries

Can we find an efficient implementation for dictionaries?

Attempt 1: Arrays

- search: $O(n)$
  - entire array must be traversed
- insertion, deletion: $O(n)$
  - Array may need to be resized (requires copying all elements to a new array

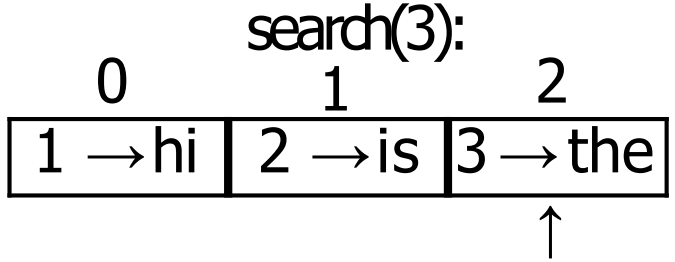


Implementations of Dictionaries

Can we find an efficient implementation for dictionaries?

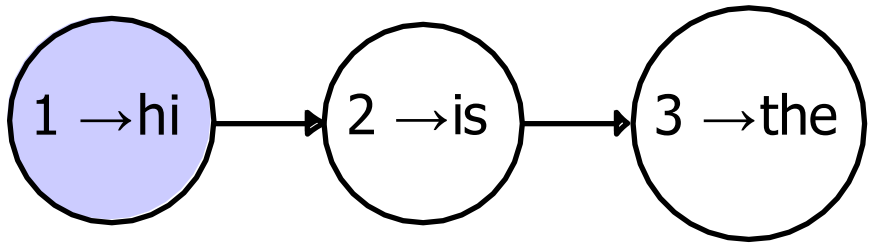
Attempt 1: Arrays

- search: $O(n)$
  - entire array must be traversed
- insertion, deletion: $O(n)$
  - Array may need to be resized (requires copying all elements to a new array



Attempt 2: Linked Lists

- search, deletion: $O(n)$
  - Entire list must be traversed
- insertion: $O(1)$
  - Can easily insert at the front of the list

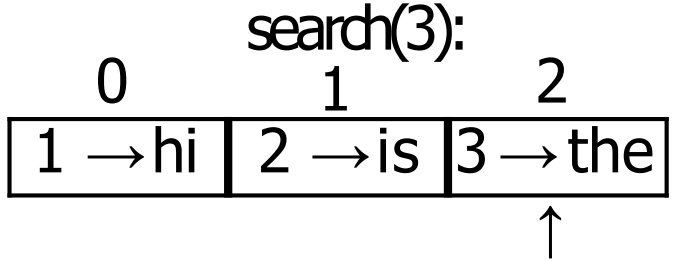


Implementations of Dictionaries

Can we find an efficient implementation for dictionaries?

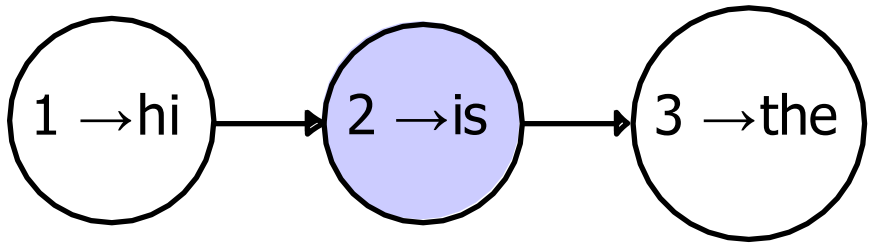
Attempt 1: Arrays

- search: $O(n)$
  - entire array must be traversed
- insertion, deletion: $O(n)$
  - Array may need to be resized (requires copying all elements to a new array



Attempt 2: Linked Lists

- search, deletion: $O(n)$
  - Entire list must be traversed
- insertion: $O(1)$
  - Can easily insert at the front of the list

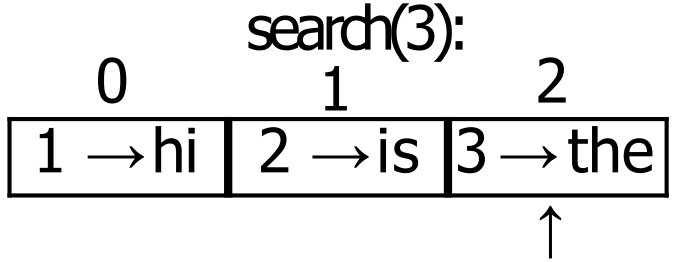


Implementations of Dictionaries

Can we find an efficient implementation for dictionaries?

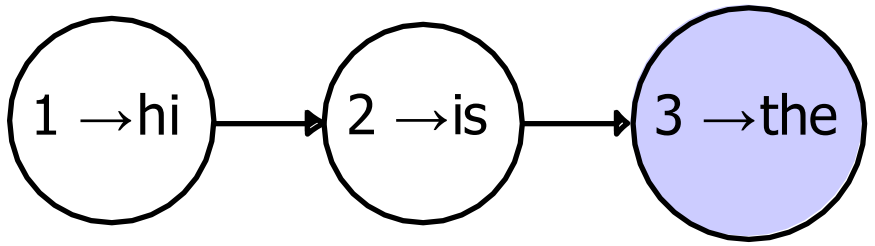
Attempt 1: Arrays

- search: $O(n)$
  - entire array must be traversed
- insertion, deletion: $O(n)$
  - Array may need to be resized (requires copying all elements to a new array



Attempt 2: Linked Lists

- search, deletion: $O(n)$
  - Entire list must be traversed
- insertion: $O(1)$
  - Can easily insert at the front of the list



Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree

Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.
  search tree property:
  - All keys to the left of a node are < that node's key
  - All keys to the right of a node are > that node's key
  - The left and right subtrees of the node also satisfy the search tree property

Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.
  search tree property:

- All keys to the left of a node are < that node's key
- All keys to the right of a node are > that node's key
- The left and right subtrees of the node also satisfy the search tree property

insert (4, "a")

Dictionaries: Binary Search Trees

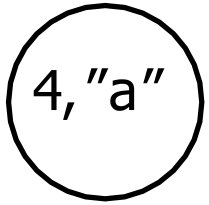
Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.

search tree property:

- All keys to the left of a node are < that node's key
- All keys to the right of a node are > that node's key
- The left and right subtrees of the node also satisfy the search tree property

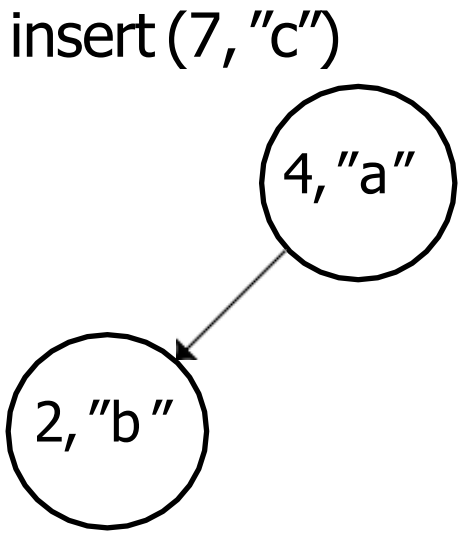
insert (2, "b")



Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

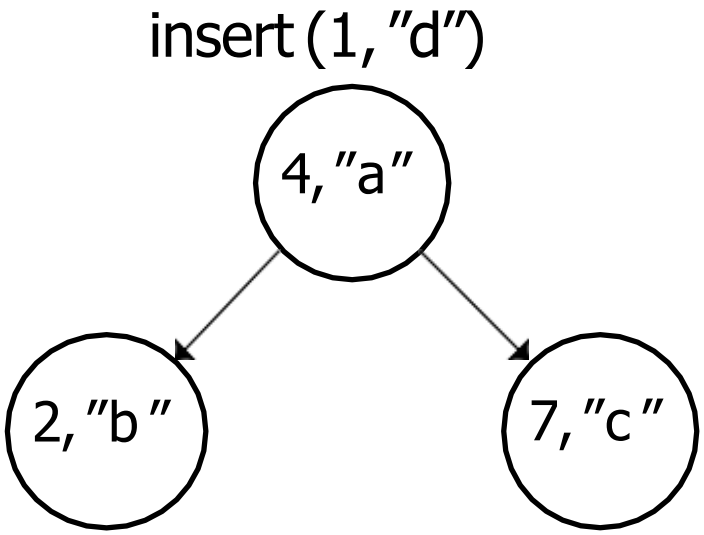
- Store items in nodes of a binary tree
  - Keys in the tree are ordered. **search tree property:**
    - All keys to the left of a node are < that node's key
    - All keys to the right of a node are > that node's key
    - The left and right subtrees of the node also satisfy the search tree property



Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

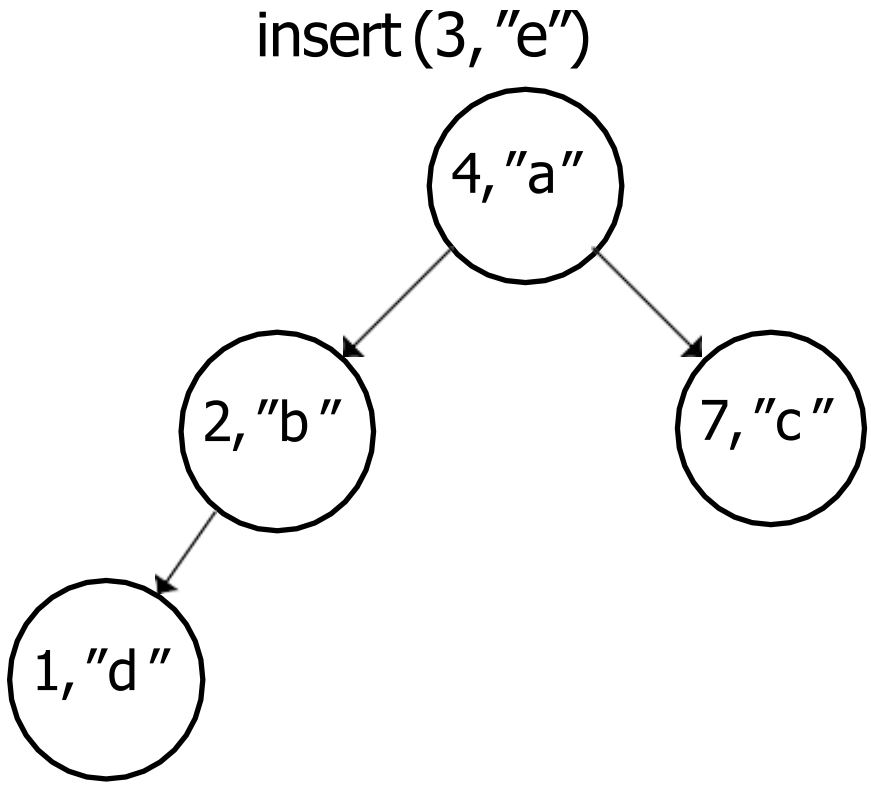
- Store items in nodes of a binary tree
- Keys in the tree are ordered. **search tree property:**
  - All keys to the left of a node are < that node's key
  - All keys to the right of a node are > that node's key
  - The left and right subtrees of the node also satisfy the search tree property



Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.
  search tree property:
  - All keys to the left of a node are < that node's key
  - All keys to the right of a node are > that node's key
  - The left and right subtrees of the node also satisfy the search tree property

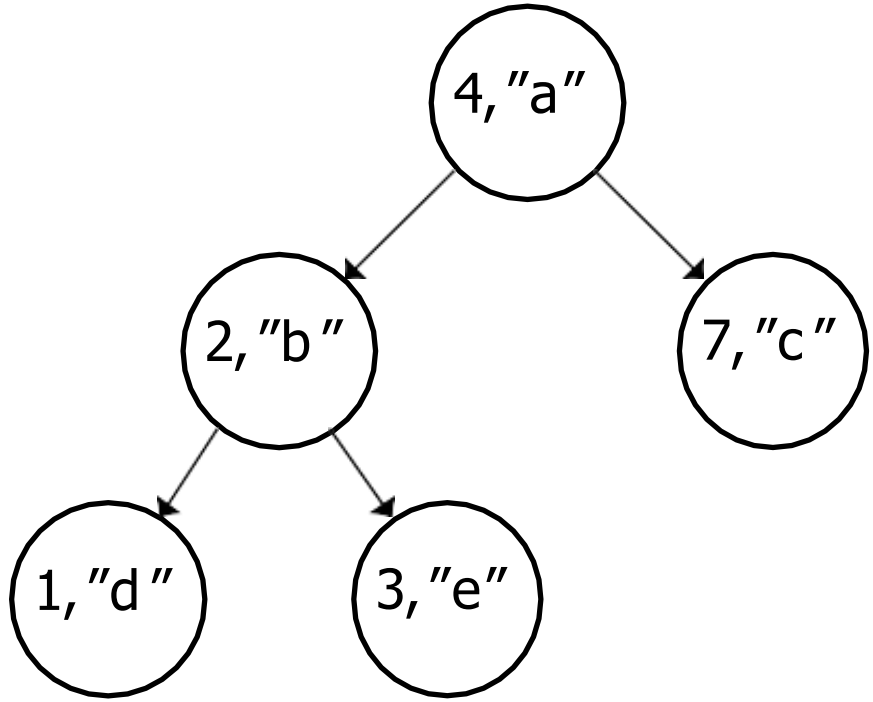


Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.
  search tree property:

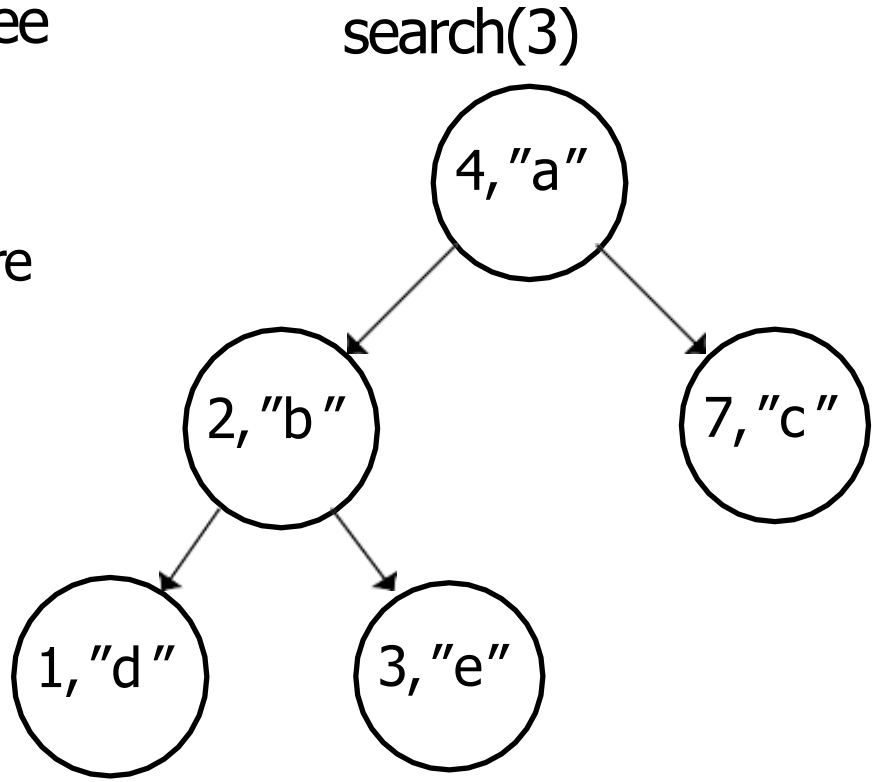
- All keys to the left of a node are < that node's key
- All keys to the right of a node are > that node's key
- The left and right subtrees of the node also satisfy the search tree property



Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.
  search tree property:
  - All keys to the left of a node are < that node's key
  - All keys to the right of a node are > that node's key
  - The left and right subtrees of the node also satisfy the search tree property



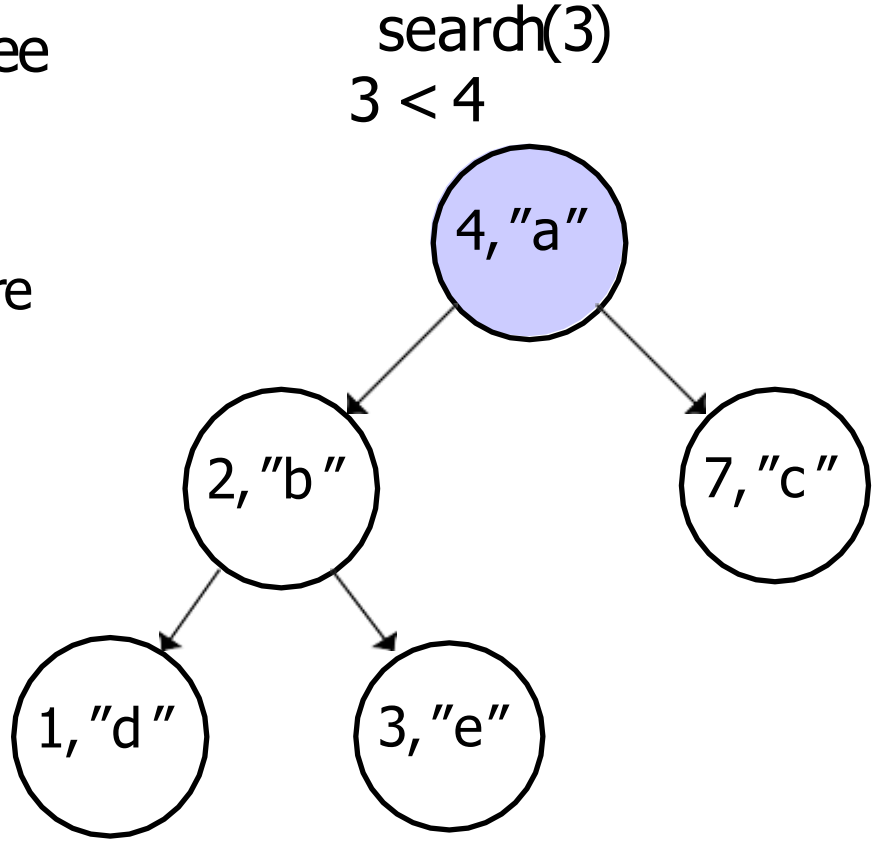
Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.

  search tree property:

  - All keys to the left of a node are < that node's key
  - All keys to the right of a node are > that node's key
  - The left and right subtrees of the node also satisfy the search tree property



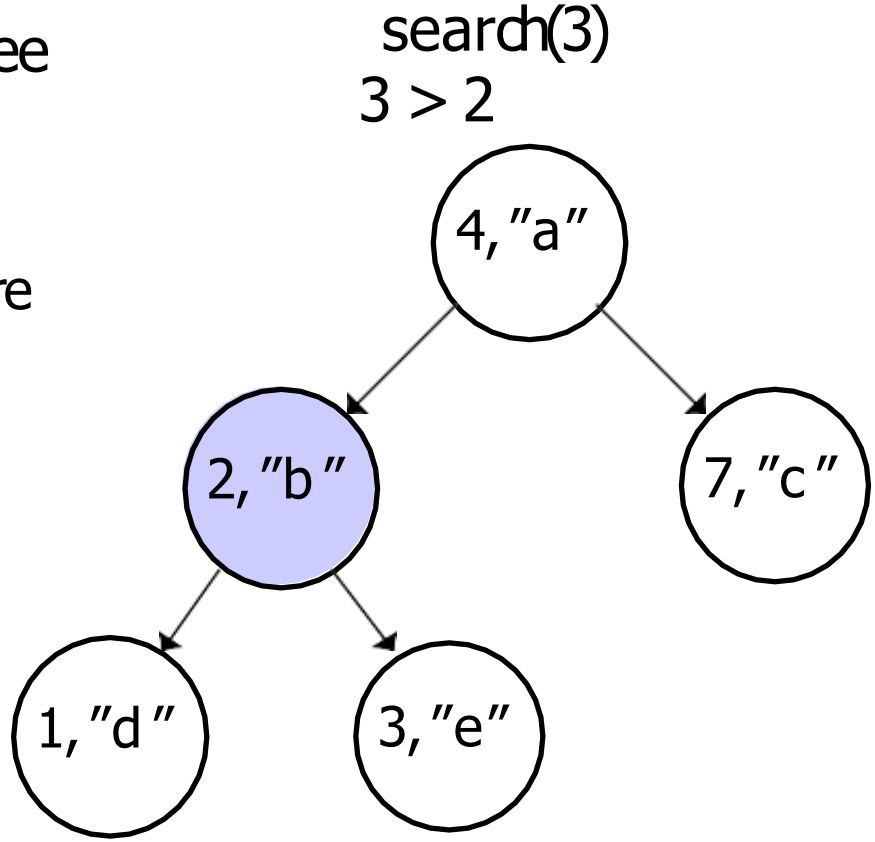
Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.

  search tree property:

  - All keys to the left of a node are < that node's key
  - All keys to the right of a node are > that node's key
  - The left and right subtrees of the node also satisfy the search tree property

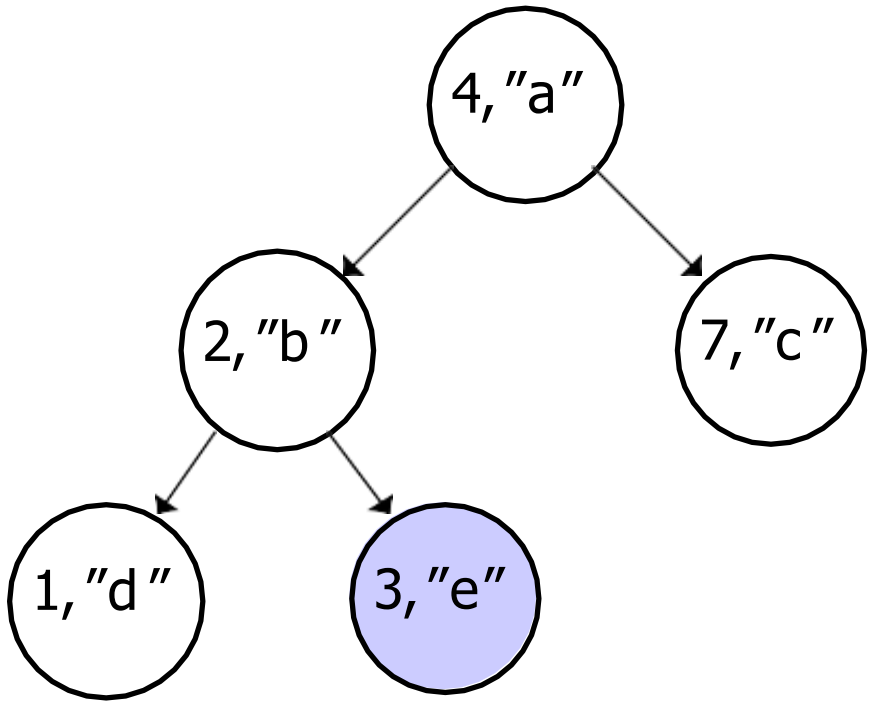


Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.
  search tree property:
  - All keys to the left of a node are < that node's key
  - All keys to the right of a node are > that node's key
  - The left and right subtrees of the node also satisfy the search tree property

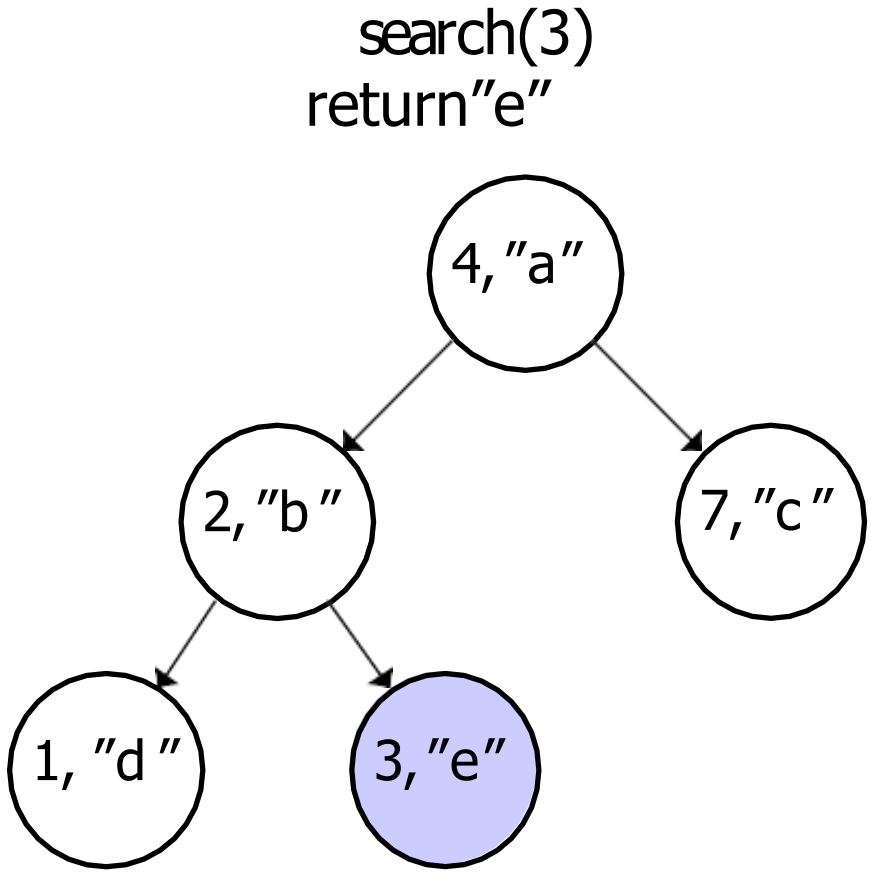
search(3)
return "e"



Dictionaries: Binary Search Trees

Attempt 3: Binary Search Tree

- Store items in nodes of a binary tree
- Keys in the tree are ordered.
  search tree property:
  - All keys to the left of a node are < that node's key
  - All keys to the right of a node are > that node's key
  - The left and right subtrees of the node also satisfy the search tree property



Time to insert, search and delete is proportional to the height of the tree!

Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree

Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?

Binary Search Trees: Runtime

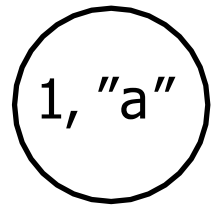
- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?

insert(1, "a")

Binary Search Trees: Runtime

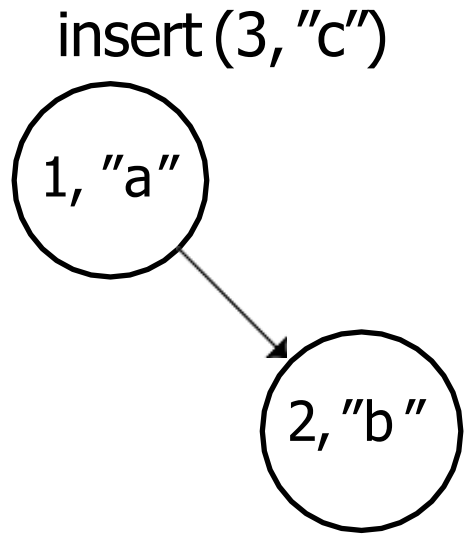
- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?

insert (2, "b")



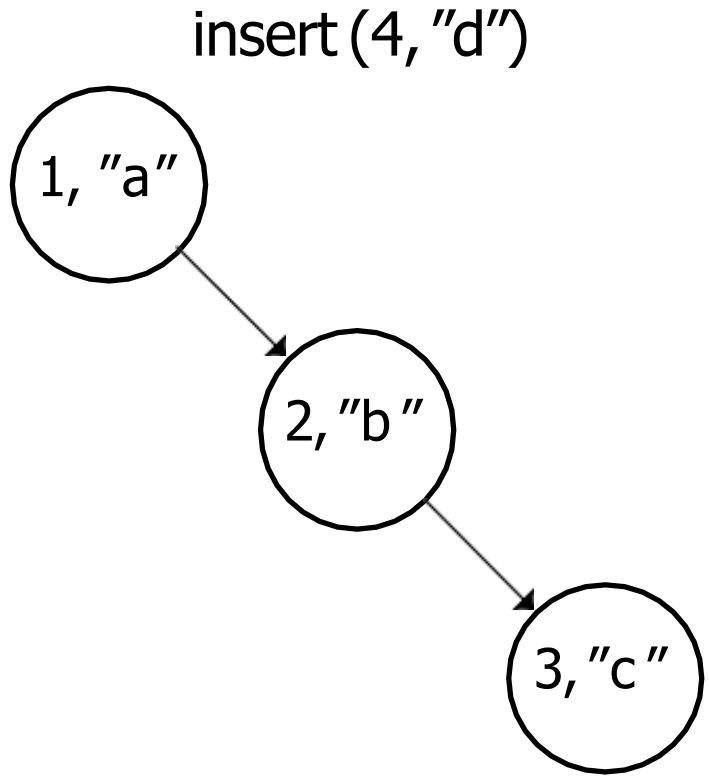
Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?



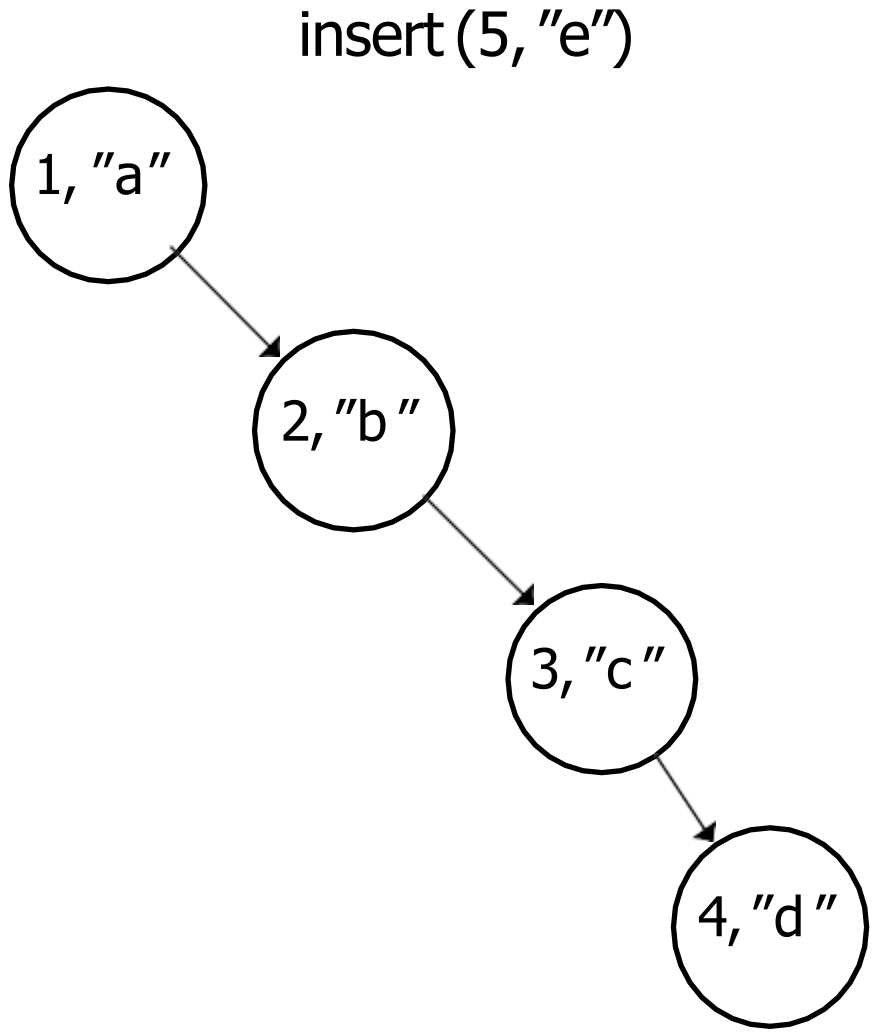
Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?



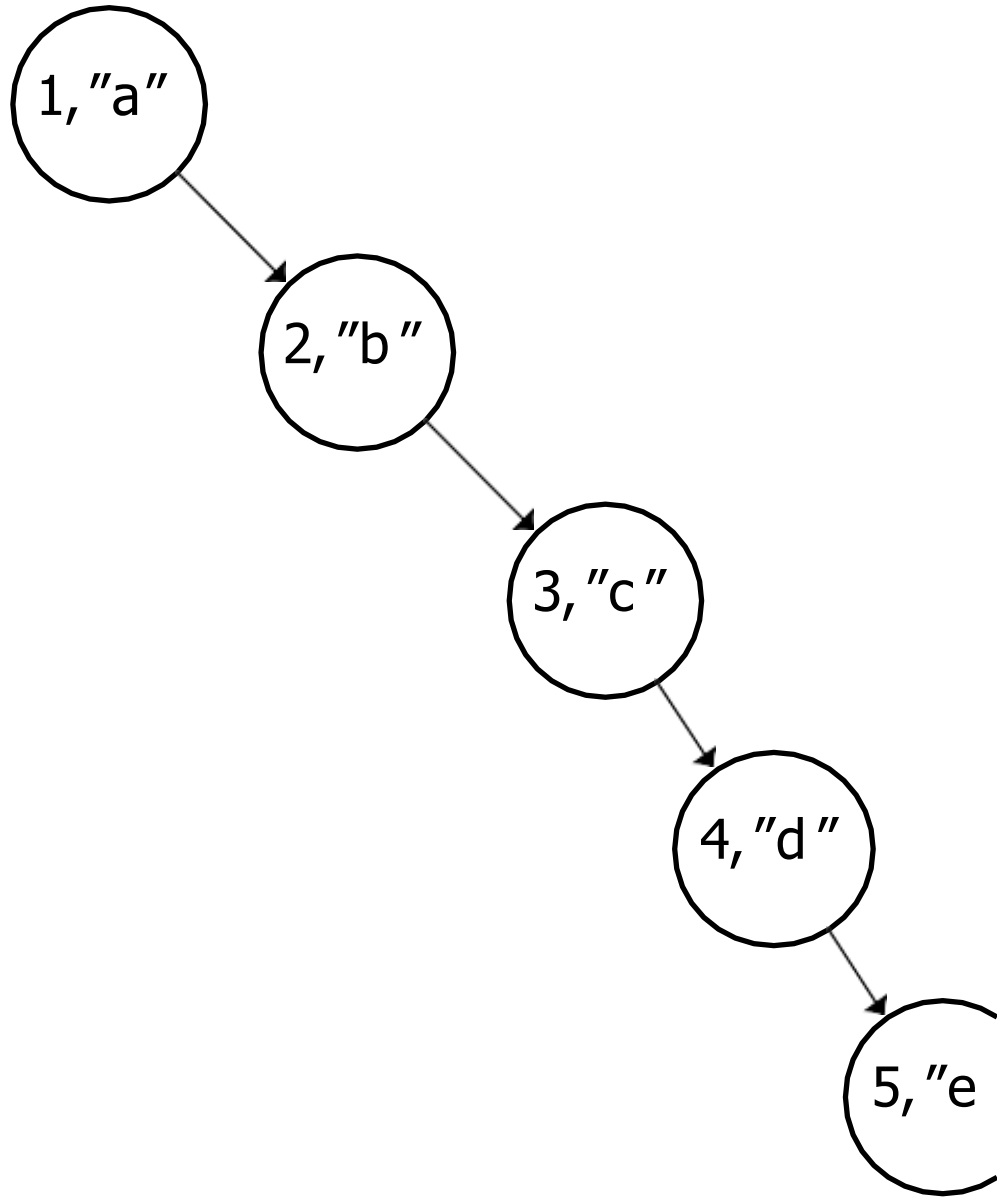
Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?



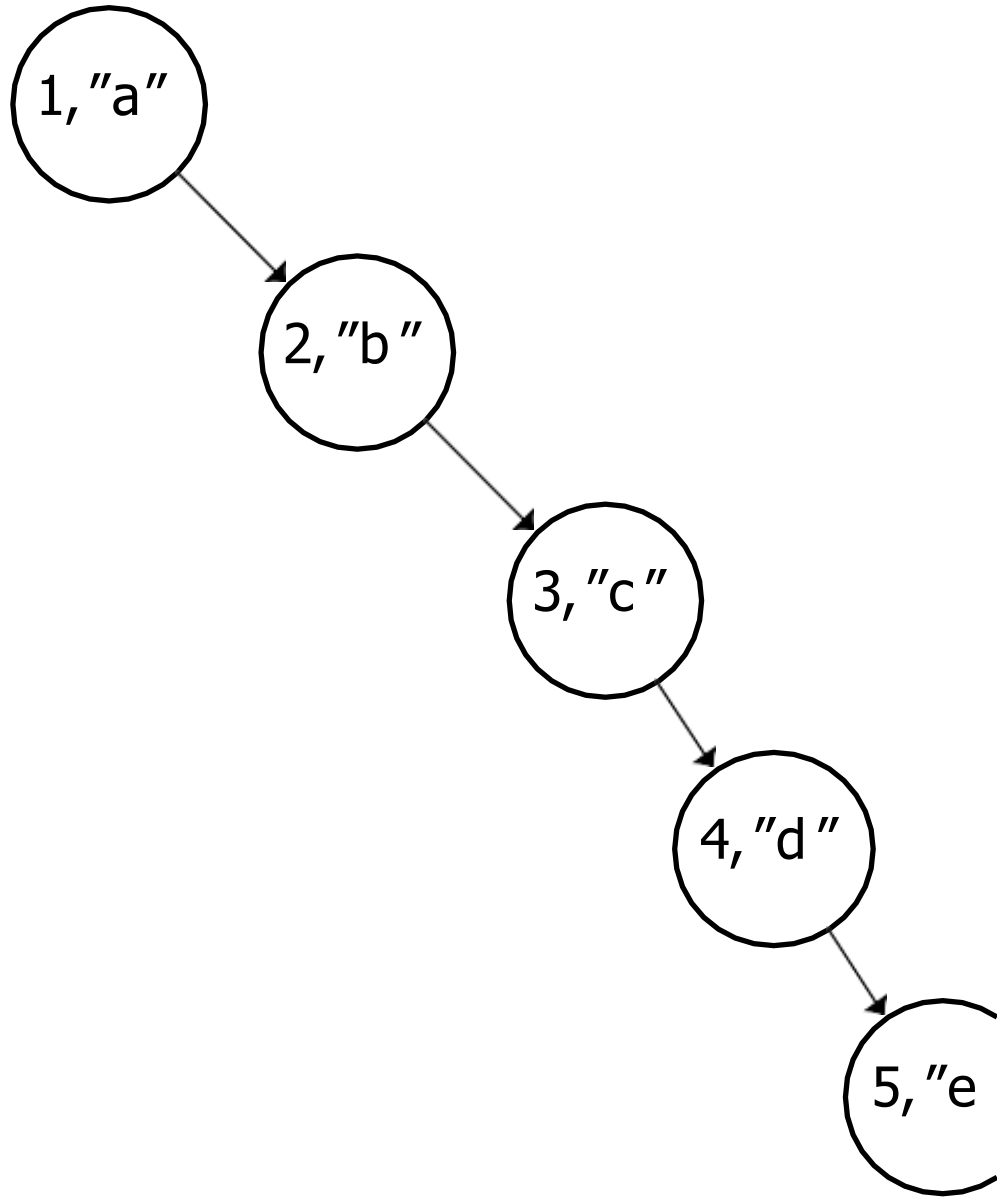
Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?



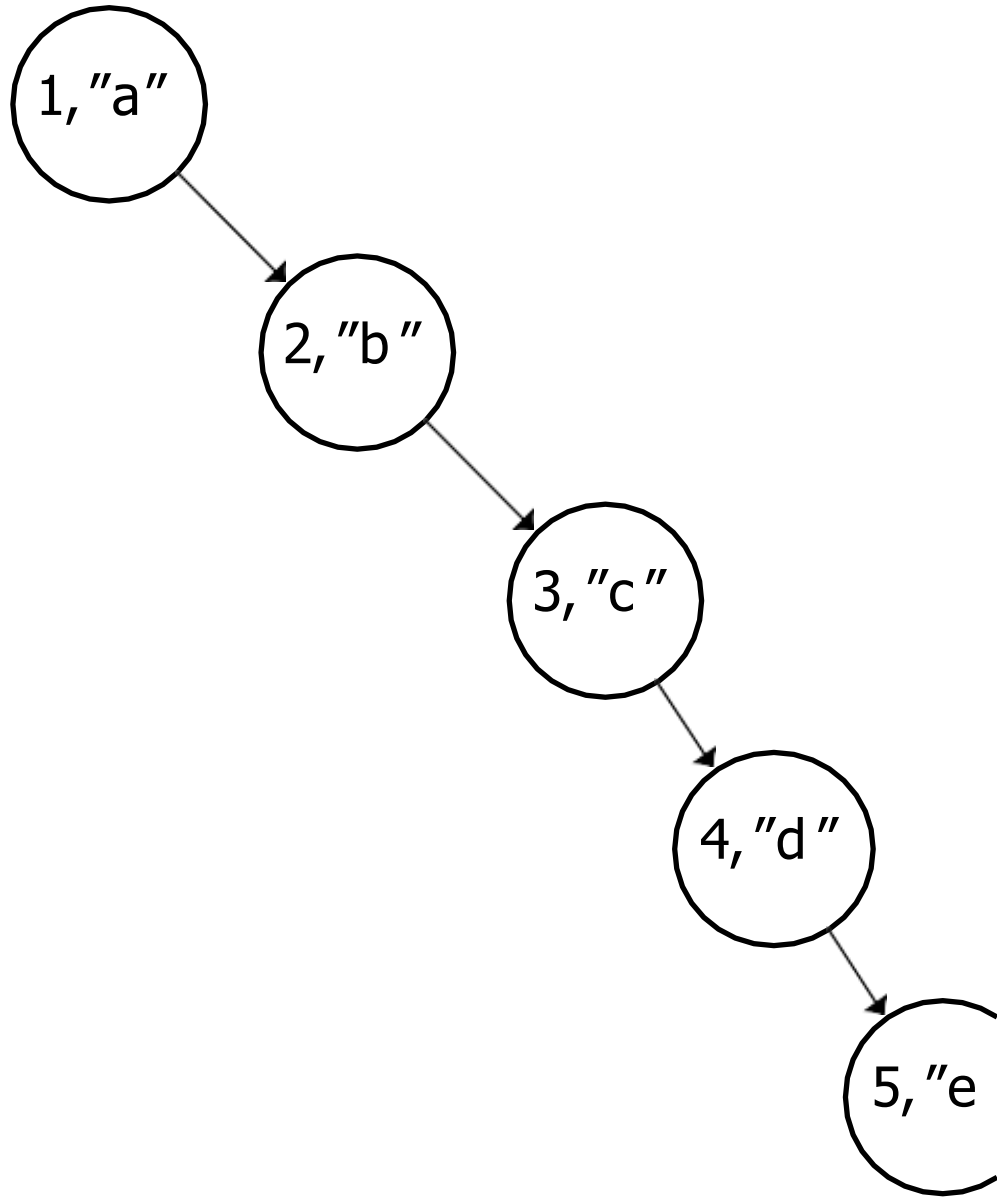
Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?
- Worst case: height of tree is $O(n)$ (number of elements inserted)



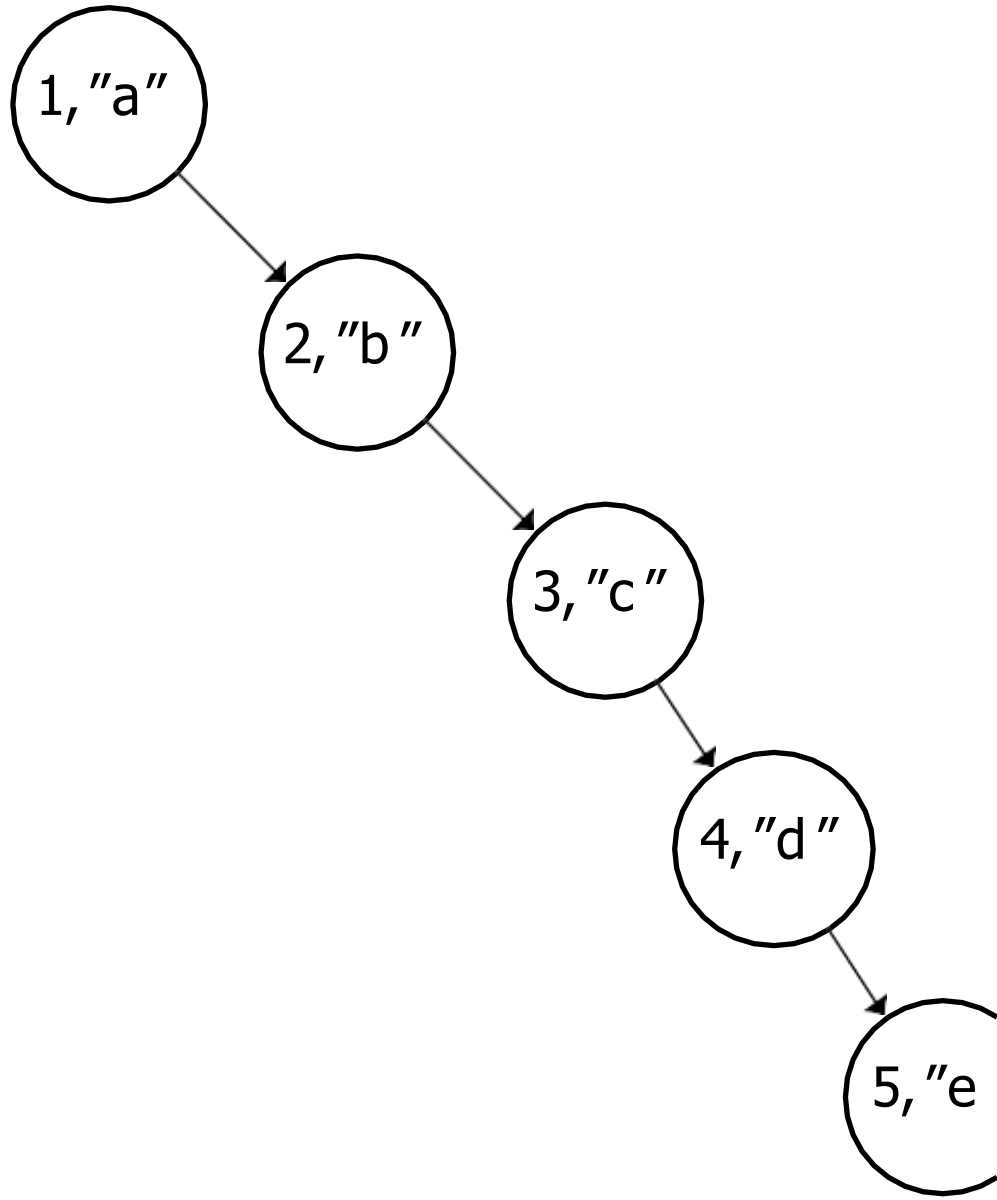
Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?
- Worst case: height of tree is $O(n)$ (number of elements inserted)
- However, common case: tree is balanced.



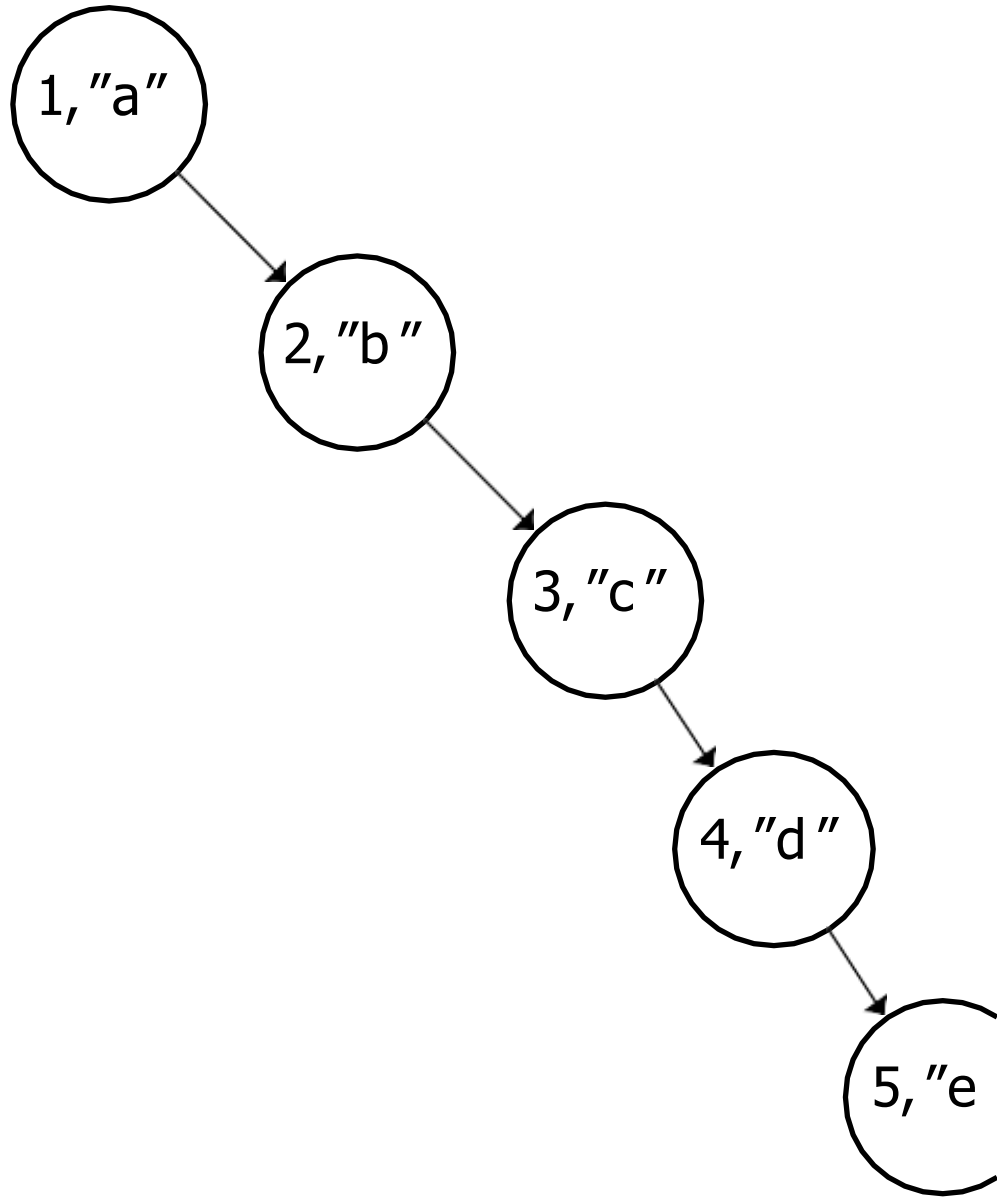
Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?
- Worst case: height of tree is $O(n)$ (number of elements inserted)
- However, common case: tree is balanced.
  - 1st level: 1 node
  - 2nd level: 2 nodes
  - kth level: 2^k nodes
  - $n = 1 + 2 + 2^2 + \ldots + 2^k$
  - $2^{k+1} - 1 = n$, $k = O(lgn)$



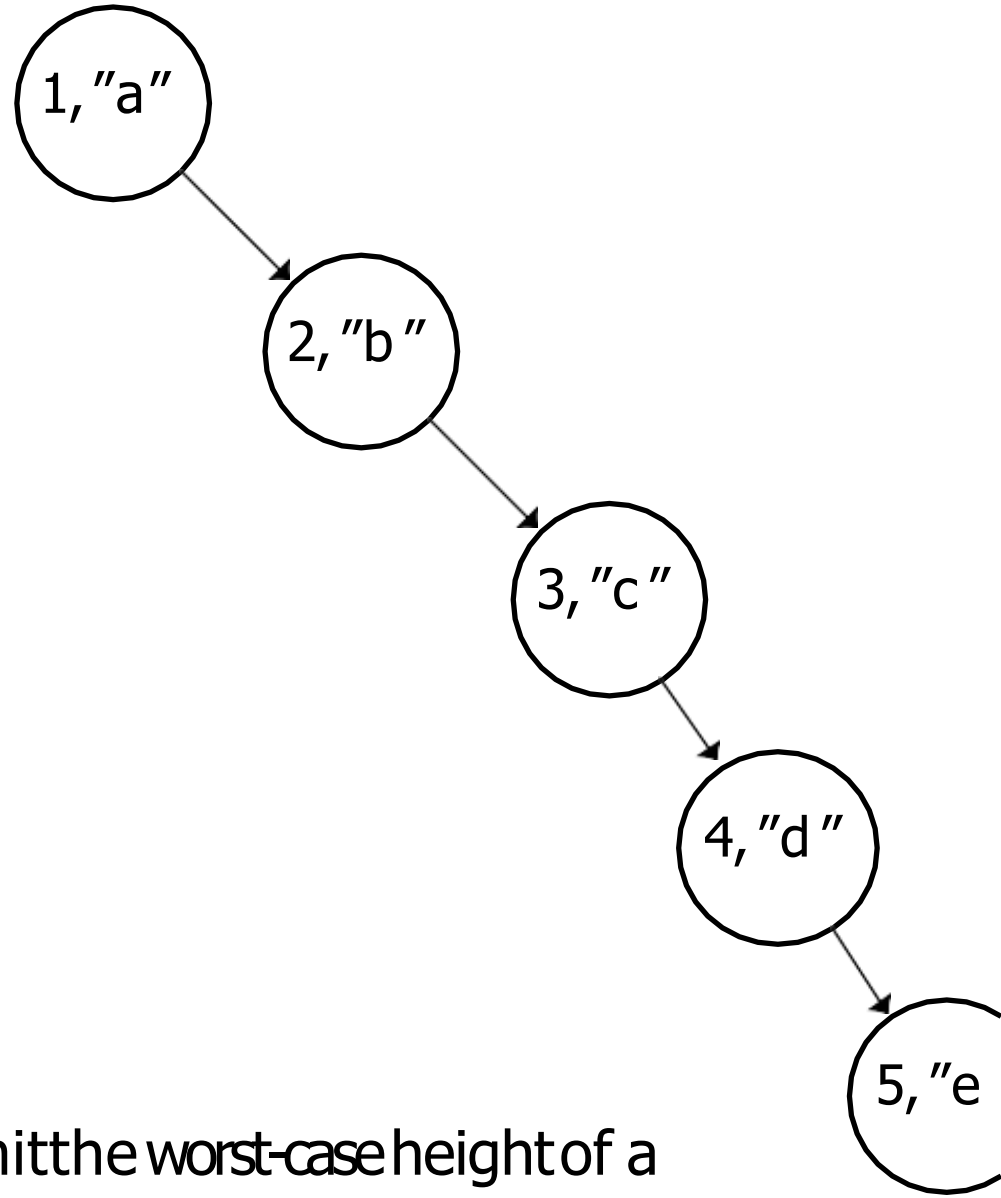
Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?
- Worst case: height of tree is $O(n)$ (number of elements inserted)
- However, common case: tree is balanced.
  - 1st level: 1 node
  - 2nd level: 2 nodes
  - kth level: 2^k nodes
  - $n = 1 + 2 + 2^2 + \ldots + 2^k$
  - $2^{k+1} - 1 = n$, $k = O(lgn)$
- common case: height is $O(lgn)$



Binary Search Trees: Runtime

- Insert, Deletion and Search take time proportional to height of the tree
- But how bad can the height be?
- Worst case: height of tree is $O(n)$ (number of elements inserted)
- However, common case: tree is balanced.
  - 1st level: 1 node
  - 2nd level: 2 nodes
  - kth level: 2^k nodes
  - $n = 1 + 2 + 2^2 + \ldots + 2^k$
  - $2^{k+1} - 1 = n$, $k = O(lgn)$
- common case: height is $O(lgn)$



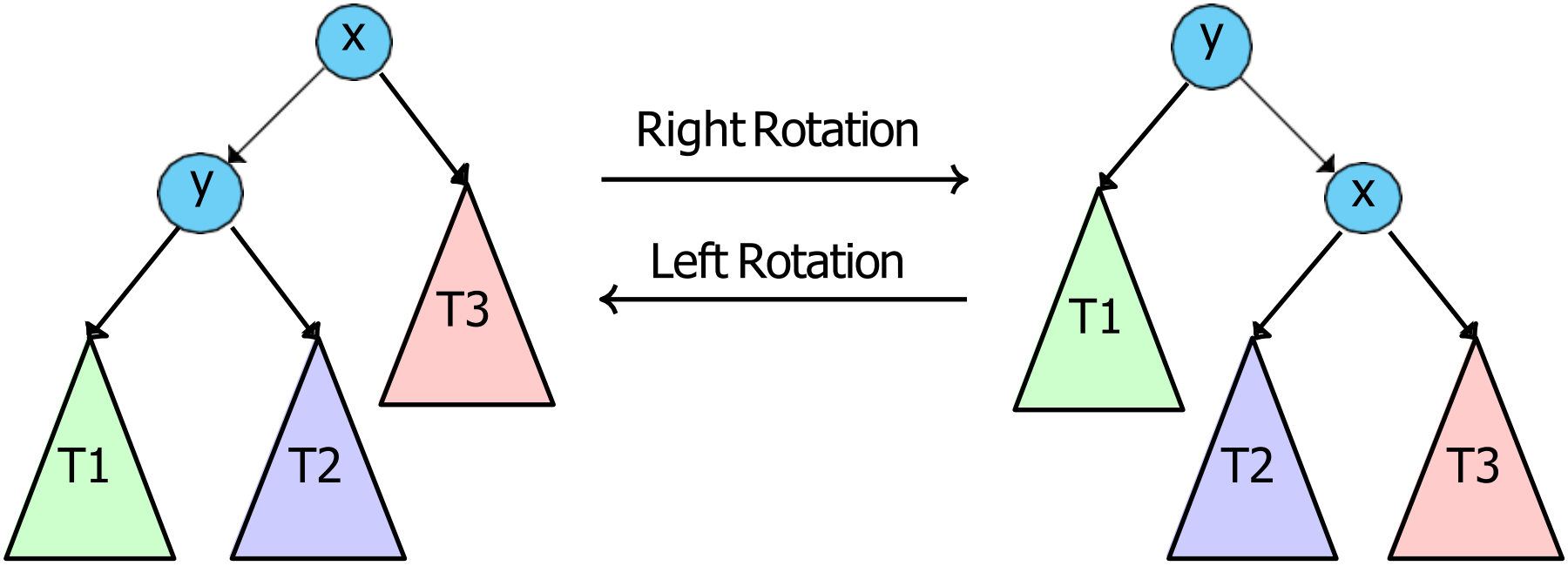
Is there anything we can do to limit the worst-case height of a binary search tree?

Video 1.9
Sampath Kannan

Balanced Binary Search Trees

- BSTs can become unbalanced leading to $O(n)$ run times for operations.
- We need a way to modify them so that their height is $O(\log n)$ instead of $O(n)$.
- Intuitively we can get this property if the left and right sub-trees always have similar heights
- Modifications must preserve search tree property

Rotations



We use rotations to keep left and right sub-trees balanced. In an AVL tree we maintain the invariant that all left and right sub-trees have a height difference of at most 1.

Hashing

- To use an array to implement a dictionary we need a way to map elements from our universe to indices. This mapping is called a **hash function** and the array is called a **hash table**
- Example: If our universe is all the integers and we have a hash table of size 37 we could use $h(x) = x \bmod 37$ as our hash function.
- If only one item gets mapped to each index then all operations are $O(1)$!

Load factor

- Suppose we have m different keys and a hash table of size n, and suppose that for each key we randomly choose an index to map it to.

Load factor

- Suppose we have m different keys and a hash table of size n, and suppose that for each key we randomly choose an index to map it to.
- $P(h(k) = i) = 1/n$.

Load factor

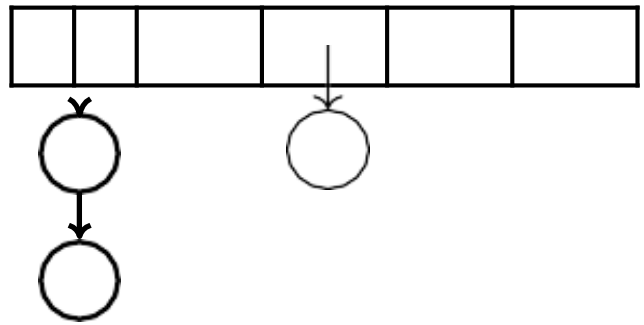
- Suppose we have m different keys and a hash table of size n, and suppose that for each key we randomly choose an index to map it to.
- $P(h(k) = i) = 1/n$.
- Let X_i be the number of keys mapped to index i and

$$E[X] = \sum_k P(h(k) = i) * (1) = \sum_k (1/n) * (1) = m/n$$

- **load factor** = α.

Handling Collisions

- Can't get rid of collisions so we need to store multiple items in a single bin
- One approach to this is **chaining**:



- Instead of storing each item directly in the array, we store a linked list of all the items that map to that index
- Run-time of all operations is now proportional to the length of the linked lists at the index we are operating on. We just saw that this gives *expected* $O(a)$ performance.
- Note that the worst case is still $O(m)$!

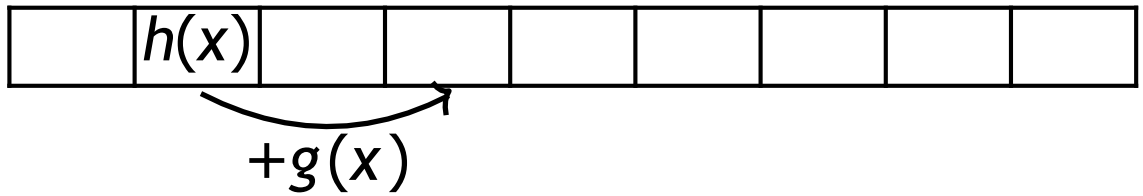
Handling Collisions 2

- Instead of chaining we can use **open addressing** where keys that map to the same index are stored in separate locations in the table.
- One approach to this is **double hashing**, where we use 2 hash functions $h(x)$ and $g(x)$.
- When there is a collision at $h(x)$ we try to insert at $h(x) + g(x)$, then $h(x) + 2g(x)$, ... etc

- Pros: No extra storage required, we don't have to deal with pointers
- Cons: Deletion is very tricky and easy to mess up

Handling Collisions 2

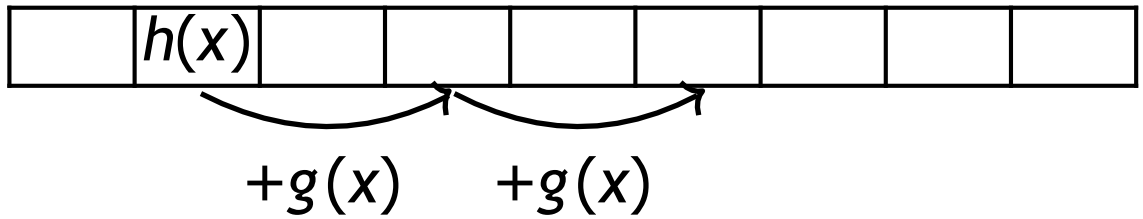
- Instead of chaining we can use **open addressing** where keys that map to the same index are stored in separate locations in the table.
- One approach to this is **double hashing**, where we use 2 hash functions $h(x)$ and $g(x)$.
- When there is a collision at $h(x)$ we try to insert at $h(x) + g(x)$, then $h(x) + 2g(x)$, ... etc



- Pros: No extra storage required, we don't have to deal with pointers
- Cons: Deletion is very tricky and easy to mess up

Handling Collisions 2

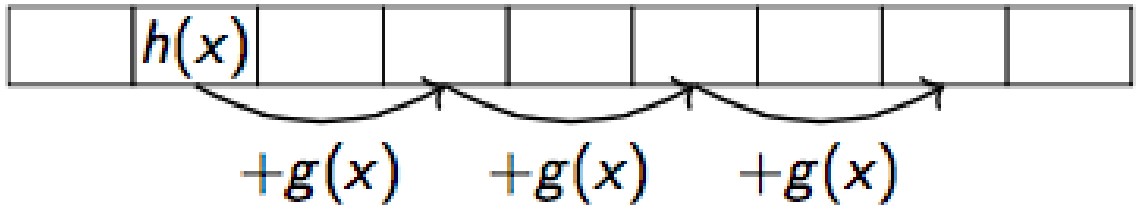
- Instead of chaining we can use **open addressing** where keys that map to the same index are stored in separate locations in the table.
- One approach to this is **double hashing**, where we use 2 hash functions $h(x)$ and $g(x)$.
- When there is a collision at $h(x)$ we try to insert at $h(x) + g(x)$, then $h(x) + 2g(x)$, ... etc



- Pros: No extra storage required, we don't have to deal with pointers
- Cons: Deletion is very tricky and easy to mess up

Handling Collisions 2

- Instead of chaining we can use **open addressing** where keys that map to the same index are stored in separate locations in the table.
- One approach to this is **double hashing,** where we use 2 hash functions $h(x)$ and $g(x)$.
- When there is a collision at $h(x)$ we try to insert at $h(x) + g(x)$, then $h(x) + 2g(x)$, ... etc



- Pros: No extra storage required, we don't have to deal with pointers
- Cons: Deletion is very tricky and easy to mess up