# CSE 332

## JUNE 23RD – PRIORITY QUEUES AND THE HEAP

# TODAY'S LECTURE

- **Priority Queue ADT**
- **Heap DS**
  - Heap Property
  - Completeness property
- **Implementation**

# REVIEW FROM LAST WEEK

- Priority Queue

# REVIEW FROM LAST WEEK

- **Priority Queue**
  - Data enqueued with a priority

# REVIEW FROM LAST WEEK

- **Priority Queue**
  - Data enqueued with a priority
  - Lower priority data dequeue first

# REVIEW FROM LAST WEEK

- **Priority Queue**
  - Data enqueued with a priority
  - Lower priority data dequeue first
  - Maintain queue principle?

# REVIEW FROM LAST WEEK

- **Priority Queue**
  - Data enqueued with a priority
  - Lower priority data dequeue first
  - Maintain queue principle?
- **Implementations?**

# REVIEW FROM LAST WEEK

- **Priority Queue**
  - Data enqueued with a priority
  - Lower priority data dequeue first
  - Maintain queue principle?
- **Implementations?**
  - Array and Linked List both have serious flaws.

# HEAP-ORDER PROPERTY

- Still a binary tree
- Instead of search (left < parent),

# HEAP-ORDER PROPERTY

- Still a binary tree
- Instead of search (left < parent),
  parent should be less than children

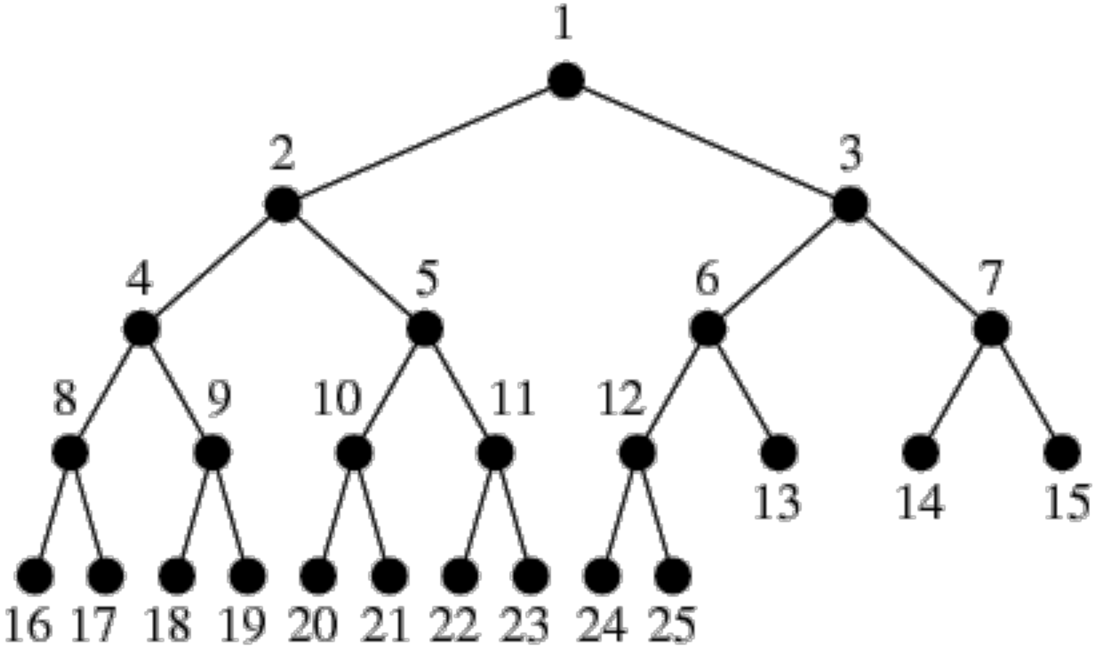
# HEAP-ORDER PROPERTY

- Still a binary tree
- Instead of search (left < parent),
  parent should be less than children
- How to implement?
- Insert and delete are different than BST

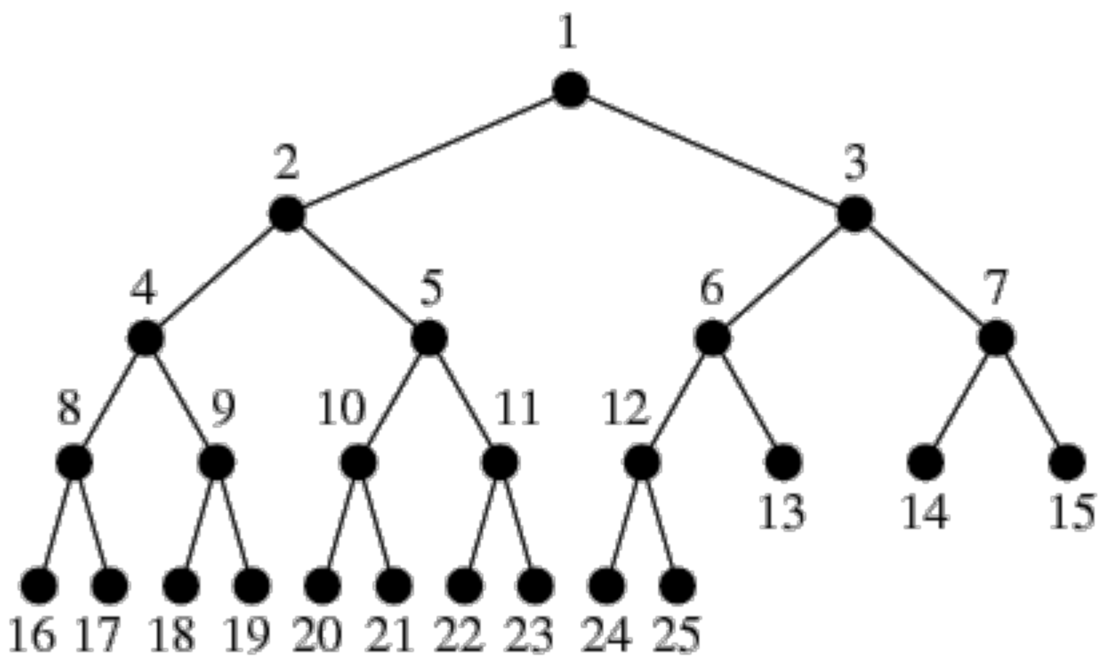
# HEAP-ORDER PROPERTY

- Still a binary tree
- Instead of search (left < parent),
  parent should be less than children
- How to implement?
- Insert and delete are different than BST

# COMPLETENESS



# COMPLETENESS




**Filling left to right and top to bottom is another property - completeness**

# HEAPS

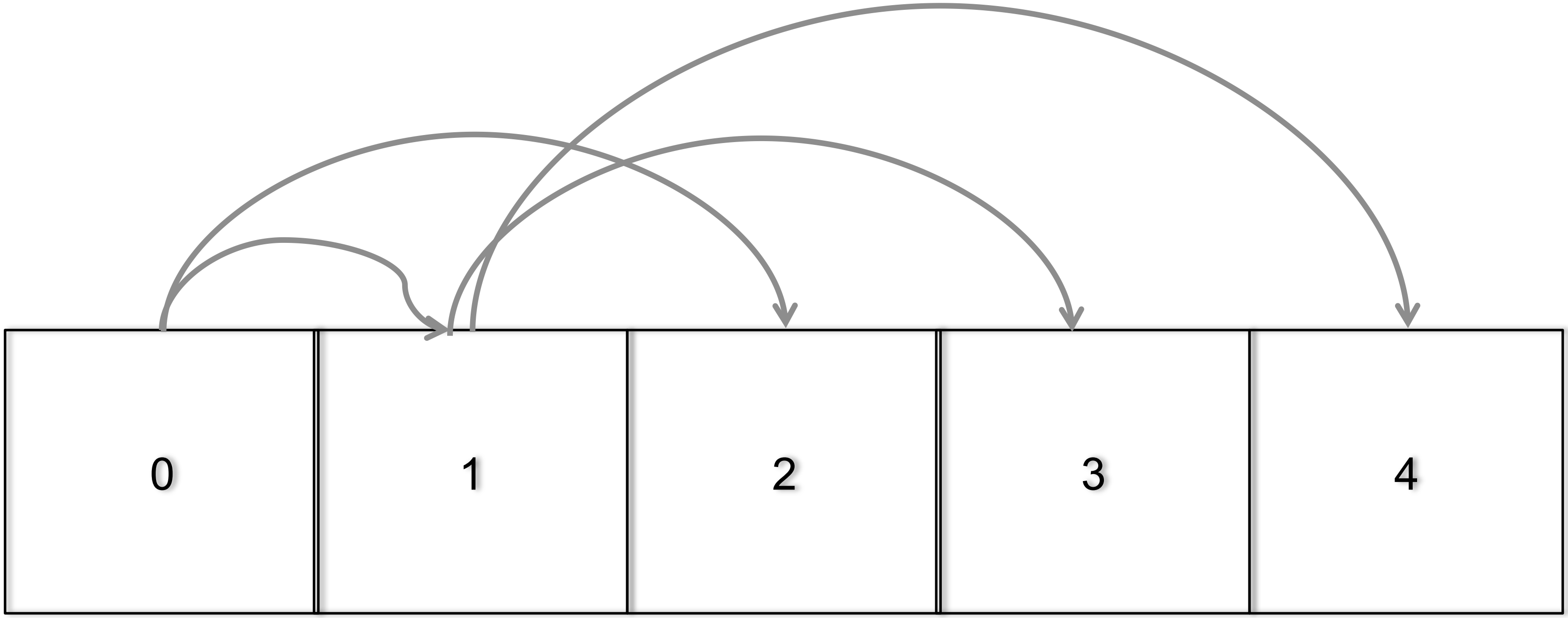
- Heap property (parents < children)
- Complete tree property (left to right, bottom to top)
- How does this help?

# HEAPS

- **Heap property (parents < children)**
- **Complete tree property (left to right, bottom to top)**
- **How does this help?**
  - Array implementation

# HEAPS

- Insert into array from left to right
- For any parent at index i,
  children at 2*i+1 and 2*i+2



# HEAPS

- How to maintain heap property then?

# HEAPS

- **How to maintain heap property then?**
  - Parent must be higher priority than children

# HEAPS

- **How to maintain heap property then?**
  - Parent must be higher priority than children
- **Two functions – percolate up and percolate down**

# SWAPPING IN THE HEAP

- **Percolate up**
  - When a new item is inserted:
    - Place the item at the next position to preserve completeness
    - Swap the item up the tree until it is larger than its parent

# SWAPPING IN THE HEAP

- **Percolate down**
  - When an item is deleted:
    - Remove the root of the tree (to be returned)
    - Move the last object in the tree to the root
    - Swap the moved piece down while it is larger than it's smallest child
    - Only swap with the smallest child

# HEAPS AS ARRAYS

- **Because heaps are complete, they can be represented as arrays without any gaps in them.**
- **Naïve implementation:**
    - Left child: 2*i+1
    - Right child: 2*i + 2
    - Parent: (i-1)/2

# HEAPS AS ARRAYS

- **Alternate (common) implementation:**
  - Put the root of the array at index 1
  - Leave index 0 blank
  - Calculating children/parent becomes:
    - Left child: 2*i
    - Right child: 2*i + 1
    - Parent: i/2

# HEAPS AS ARRAYS

- Why do an array at all?

# HEAPS AS ARRAYS

- **Why do an array at all?**
  - + Memory efficiency
  - + Fast accesses to data
  - + Forces log n depth
  - - Needs to resize
  - - Can waste space

# HEAPS AS ARRAYS

- **Why do an array at all?**
  - + Memory efficiency
  - + Fast accesses to data
  - + Forces log n depth
  - - Needs to resize
  - - Can waste space
- **Overall, however, better done through an array**

# ALGORITHM ANALYSIS

- Important topic. Why?

# ALGORITHM ANALYSIS

- **Important topic. Why?**
    - Show that an implementation is better.

# ALGORITHM ANALYSIS

- **Important topic. Why?**
  - Show that an implementation is better.
- **What do we mean by better?**

# ALGORITHM ANALYSIS

- **Important topic. Why?**
  - Show that an implementation is better.
- **What do we mean by better?**
  - Fewer clock cycles
  - More efficient memory usage
  - Correctness

# ALGORITHM ANALYSIS

- **Math review**
- **Logarithms**
  - $\log_2 x = y$ when $x = 2^y$

# ALGORITHM ANALYSIS

- **Math review**
- **Logarithms**
  - $\log_2 x = y$ when $x = 2^y$
  - How does this grow?

# ALGORITHM ANALYSIS

- **Math review**
- **Logarithms**
  - `log`$_2$ `x = y` when `x = 2`$^y$
  - How does this grow? Slowly
  - A balanced tree has a height ~$\log_2$ n
  - `log`$_k$ `x` differs from `log`$_j$ `x` by a constant factor

# ALGORITHM ANALYSIS

- **Operations**
  - `log(A*B) = log(A) + log(B)`
  - `log(A/B) = log(A) – log(B)`
  - $\log(A^B) = B * \log(A)$

# ALGORITHM ANALYSIS

- Floor and ceiling

# ALGORITHM ANALYSIS

- **Floor and ceiling**
  - Integer rounding, computers operate in integer quantities
    - Clock cycles
    - Memory bytes

# ALGORITHM ANALYSIS

- **Floor and ceiling**
  - Integer rounding, computers operate in integer quantities
    - Clock cycles
    - Memory bytes

**Floor :** $\lfloor X \rfloor$ **denotes largest integer $\leq$ x**

**Ceiling:** $\lceil X \rceil$ **denotes smallest integer $\geq$ x**

# ALGORITHM ANALYSIS

- Operations

# ALGORITHM ANALYSIS

- **Operations**
  - Arithmetic
  - Comparisons
  - Memory reads/writes
- **Loops and functions are just chains of these operations.**

# ALGORITHM ANALYSIS

```
Int value = 0;
for(int i = 0; i < 10; i ++){
      value++;
}
```

# ALGORITHM ANALYSIS

```
Int value = 0;
for(int i = 0; i < 10; i ++){
      value++;
}
```

**How long does this take?**

# ALGORITHM ANALYSIS

```
Int value = 0;
for(int i = 0; i < 10; i ++){
      value++;
}
```

**How long does this take?**

**How many operations?**

# ALGORITHM ANALYSIS

```
Int value = 0; 1
for(int i = 0; i < 10; i ++){ 10
      value++;    1
}
```

**How long does this take?**

**How many operations?**

# ALGORITHM ANALYSIS

```
Int value = 0; 1 + 1
for(int i = 0; i < 10; i ++){ 10
      value++;    1
}
```

**How long does this take?**

**How many operations?**

# ALGORITHM ANALYSIS

```
Int value = 0; 1 + 1
for(int i = 0; i < 10; i ++){ 10
      value++;    1
}
```

**How long does this take?**

**How many operations?**

**2+11+10 = 23**

# ALGORITHM ANALYSIS

```
Int value = 0;
for(int i = 0; i < N; i ++){
      value++;
}
```

**How long does this take?**

# ALGORITHM ANALYSIS

```
Int value = 0;
for(int i = 0; i < N; i ++){
    value++;
}
```

**How long does this take?**

**1+1+(N+1) + N**

# ALGORITHM ANALYSIS

- Principles of analysis

# ALGORITHM ANALYSIS

- **Principles of analysis**
  - Determining performance behavior

# ALGORITHM ANALYSIS

- **Principles of analysis**
  - Determining performance behavior
  - How does an algorithm react to new data or changes?

# ALGORITHM ANALYSIS

- **Principles of analysis**
  - Determining performance behavior
  - How does an algorithm react to new data or changes?
  - Independent of language or implementation

# ALGORITHM ANALYSIS

- Example: find()
- Suppose an array with 15 elements

# ALGORITHM ANALYSIS

- Example: find()
- Suppose an array with 15 elements
- One implementation has a sorted array,
  the other is unsorted

# ALGORITHM ANALYSIS

- Example: find()
- Suppose an array with 15 elements
- One implementation has a sorted array, the other is unsorted
- For which one will find() be faster?

# ALGORITHM ANALYSIS

- Example: find()
- Suppose an array with 5 elements
- One implementation has a sorted array, the other is unsorted
- For which one will find() be faster?
- How long will it take?

# ALGORITHM ANALYSIS

- **Find(1)**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 5 | 3 | 1 | | | |

# ALGORITHM ANALYSIS

- Find(1)
- How many operations?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 5 | 3 | 1 | | | |

# ALGORITHM ANALYSIS

- **Find(4)?**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 5 | 3 | 1 | | | |

# ALGORITHM ANALYSIS

- Not a good representation of how the algorithm actually behaves.
- Want to assess the algorithm on the whole, not just over a few inputs

# ALGORITHM ANALYSIS

- **Not a good representation of how the algorithm actually behaves.**
- **Want to assess the algorithm on the whole, not just over a few inputs**
- **This is why testing alone isn't enough**

# ALGORITHM ANALYSIS

- Possible solutions?

# ALGORITHM ANALYSIS

- **Possible solutions?**
  - Average case: find the average performance over all inputs

# ALGORITHM ANALYSIS

- **Possible solutions?**
  - Average case: find the average performance over all inputs
  - Worst case: how long the program takes to complete the worst case problems.

# ALGORITHM ANALYSIS

- **Possible solutions?**
  - Average case: can be difficult to compute

# ALGORITHM ANALYSIS

- **Possible solutions?**
  - Average case: can be difficult to compute
  - What is the average case for binary search?

# ALGORITHM ANALYSIS

- **Possible solutions?**
  - Worst case: is most commonly used

# ALGORITHM ANALYSIS

- **Possible solutions?**
  - Worst case: is most commonly used
  - Easily compared and gives a good estimate of the robustness of an algorithm

# ALGORITHM ANALYSIS

- **Possible solutions?**
  - Worst case: is most commonly used
  - Easily compared and gives a good estimate of the robustness of an algorithm

# ALGORITHM ANALYSIS

- **Worst case runtime here?**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 5 | 3 | 1 | | | |

# ALGORITHM ANALYSIS

- **Worst case runtime here?**
- **Are we convinced one is better just looking at 5 elements?**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 5 | 3 | 1 | | | |

# ASYMPTOTIC ANALYSIS

- Want to know how algorithms behave with big data

# ASYMPTOTIC ANALYSIS

- Want to know how algorithms behave with big data
- How much more does an additional element in our data structure cost us?

# ASYMPTOTIC ANALYSIS

- Consider find() for sorted v. unsorted arrays

# ASYMPTOTIC ANALYSIS

- **Consider find() for sorted v. unsorted arrays**
  - Which is better?

# ASYMPTOTIC ANALYSIS

- **Consider find() for sorted v. unsorted arrays**
  - Which is better?
  - Unsorted grows linearly – if we add one more element to the list, we expect that the algorithm will take one more operation to complete

# ASYMPTOTIC ANALYSIS

- **Consider find() for sorted v. unsorted arrays**
  - Which is better?
  - Unsorted grows linearly – if we add one more element to the list, we expect that the algorithm will take one more operation to complete
  - How much longer is an extra element in the sorted case?

# ASYMPTOTIC ANALYSIS

- **Consider find() for sorted v. unsorted arrays**
  - As trees grow exponentially in size…

# ASYMPTOTIC ANALYSIS

- **Consider find() for sorted v. unsorted arrays**
  - As trees grow exponentially in size
    they grow logarithmically in height

# ASYMPTOTIC ANALYSIS

- **Consider find() for sorted v. unsorted arrays**
  - As trees grow exponentially in size
    they grow logarithmically in height
  - Height is what determines our runtime

# ASYMPTOTIC ANALYSIS

- **Consider find() for sorted v. unsorted arrays**
  - We call the unsorted case: linear time or O(n) time
  - We call the sorted case: logarithmic time or O(log n) time

# ASYMPTOTIC ANALYSIS

- **Consider find() for sorted v. unsorted arrays**
  - We call the unsorted case: linear time or O(n) time
  - We call the sorted case: logarithmic time or O(log n) time
  - You may have seen this notation in 143

# ASYMPTOTIC ANALYSIS

- Big-O notation

# ASYMPTOTIC ANALYSIS

- **Big-O notation**
  - Captures this asymptotic behavior;
  - As we approach larger and larger elements, how long does our algorithm take to complete.

# ASYMPTOTIC ANALYSIS

- **Big-O notation**
  - Captures this asymptotic behavior;
  - As we approach larger and larger elements, how long does our algorithm take to complete.
  - Informally, if a function is `O(g(n))`, then that function grows **at most** as quickly as the function g(n)

# BINARY SEARCH

- Analyzing binary search.
- What is the worst case?

# BINARY SEARCH

- **Analyzing binary search.**
- **What is the worst case?**
  - When the item is not in the list

# BINARY SEARCH

- **Analyzing binary search.**
- **What is the worst case?**
  - When the item is not in the list
- **How long does this take to run?**

# BINARY SEARCH

- **Consider the algorithm**

```
public int binarySearch(int[] data, int toFind){
int low = 0; int high = data.length-1;
while(low <= high){
        int mid = (low+high)/2;
        if(toFind>mid) low = mid+1; continue;
        else if(toFind<mid) high = mid-1; continue;
        else return mid;
}
return -1;
}
```

# BINARY SEARCH

- **What is important here?**

# BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.

# BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**

# BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**

# BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**
  - At first iteration, N/2 elements remain

# BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**
  - At first iteration, N/2 elements remain
  - At second, N/4 elements remain

# BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**
  - At first iteration, N/2 elements remain
  - At second, N/4 elements remain
  - At the kth iteration?

# BINARY SEARCH

- **At the kth iteration:**
  - $N/2^k$ elements remain.
- **When does this terminate?**

# BINARY SEARCH

- **At the kth iteration:**
  - $N/2^k$ elements remain.
- **When does this terminate?**
  - When $N/2^k = 1$

# BINARY SEARCH

- **At the kth iteration:**
  - $N/2^k$ elements remain.
- **When does this terminate?**
  - When $N/2^k = 1$
- **How many iterations then? Solve for k.**

# BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

# BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

# BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

$\log_2 N = k$

# BINARY SEARCH

- **Solve for k.**

$N \;/\; 2^{k} = 1$

$N = 2^{k}$

$\log_2 N = k$

- **Is this exact?**

# BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

$\log_2 N = k$

- **Is this exact?**
- **Where was the error introduced?**

# BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

$\log_2 N = k$

- **Is this exact?**
- **Where was the error introduced?**
  - N can be things other than powers of two

# BINARY SEARCH

- **Solve for k.**

$N \,/\, 2^{k} = 1$

$N = 2^{k}$

$\log_2 N = k$

- **Is this exact?**
- **Where was the error introduced?**
  - N can be things other than powers of two
  - Ceiling and floor rounding

# ANALYSIS

- If this isn’t exact, is it still correct?

# ANALYSIS

- If this isn't exact, is it still correct?
- Yes. We care about asymptotic growth.

# ANALYSIS

- **If this isn't exact, is it still correct?**
- **Yes. We care about asymptotic growth.**
  - How a the runtime of an algorithm grows with big data

# ANALYSIS

- **If this isn't exact, is it still correct?**
- **Yes. We care about asymptotic growth.**
  - How a the runtime of an algorithm grows with big data
- **To incorporate this perspective, we use bigO notation**

# BIG-O NOTATION

- Informally: bigO notation denotes an upper bound for an algorithms asymptotic runtime

# BIG-O NOTATION

- Informally: bigO notation denotes an upper bound for an algorithms asymptotic runtime
- For example, if an algorithm `A` is `O(log n)`, that means some logarithmic function upper bounds `A`.

# BIG-O NOTATION

- Formally, a function `f(n)` is `O(g(n))` if there exists a `c` and $n_0$ such that:
- For all $n \geq n_0$, $f(n) < c*g(n)$
- To prove a function is O(g(n)), simply find the c and $n_0$ and demonstrate that the inequality is true

# BIG-O NOTATION

- Example: is $5n^3 + 2n$ in $O(n^4)$?

# BIG-O NOTATION

- Example: is $5n^3 + 2n$ in $O(n^4)$?
- Can we find a $c$, $n_0$ such that:
- $5n^3 + 2n \leq c*n^4$ for all $n \geq n_0$

# BIG-O NOTATION

- Example: is $5n^3 + 2n$ in $O(n^4)$?
- Can we find a $c$, $n_0$ such that:
- $5n^3 + 2n \leq c*n^4$ for all $n \geq n_0$

Let c = 7; $5n^3 + 2n \leq 7n^4$

# BIG-O NOTATION

- Example: is $5n^3 + 2n$ in $O(n^4)$?
- Can we find a $c$, $n_0$ such that:
- $5n^3 + 2n \leq c*n^4$ for all $n \geq n_0$

Let c = 7; $5n^3 + 2n \leq 7n^4$

$5n^3 + 2n \leq 5n^4 + 2n^4$

# BIG-O NOTATION

- Example: is $5n^3 + 2n$ in $O(n^4)$?
- Can we find a $c$, $n_0$ such that:
- $5n^3 + 2n \le c*n^4$ for all $n \ge n_0$

Let c = 7; $5n^3 + 2n \le 7n^4$

$5n^3 + 2n \le 5n^4 + 2n^4$

Since $n^4 \ge n^3$ and $n^4 \ge n$ for $n \ge 1$

$5n^3 + 2n \le 7n^4$ for all $n \ge 1$

# BIG-O NOTATION

- Example: is $5n^3 + 2n$ in $O(n^4)$?
- Can we find a $c$, $n_0$ such that:
- $5n^3 + 2n \leq c*n^4$ for all $n \geq n_0$

Let c = 7; $5n^3 + 2n \leq 7n^4$

$5n^3 + 2n \leq 5n^4 + 2n^4$

Since $n^4 \geq n^3$ and $n^4 \geq n$ for $n \geq 1$

$5n^3 + 2n \leq 7n^4$ for all $n \geq 1$

Therefore, $5n^3 + 2n$ is $O(n^4)$

# BIG-O NOTATION

- This is an upper bound, so if

$5n^3 + 2n$ is in $O(n^4)$, then

$5n^3 + 2n$ is in $O(n^5)$ and $O(n^n)$

# BIG-O NOTATION

- This is an upper bound, so if

$5n^3 + 2n$ is in $O(n^4)$, then

$5n^3 + 2n$ is in $O(n^5)$ and $O(n^n)$

- Is $5n^3 + 2n$ in $O(n^3)$?

# BIG-O NOTATION

- This is an upper bound, so if

$5n^3 + 2n$ is in $O(n^4)$, then

$5n^3 + 2n$ is in $O(n^5)$ and $O(n^n)$

- Is $5n^3 + 2n$ in $O(n^3)$?
- Yes, let c be 7 and $n_0 > 1$

# EXAMPLES

- $4 + 3n = O(n)$?

# EXAMPLES

- $4 + 3n = O(n)$?
- $4 + 3n = O(1)$?

# EXAMPLES

- $4 + 3n = O(n)$?
- $4 + 3n = O(1)$?
- $4 + 3n = O(n^2)$
- $n + 2 \log n = O(\log n)$?

# EXAMPLES

- $4 + 3n = O(n)$?
- $4 + 3n = O(1)$?
- $4 + 3n = O(n^2)$
- $n + 2 \log n = O(\log n)$?
- $\log n = O(n + 2 \log n)$?