CS 61B Midterm 2 Review

Dan Wang, George Yiu, Lewin Gan, and Richard Hsu

Eta Kappa Nu, Mu Chapter
University of California, Berkeley

8 April 2012

True/False

Determine the truth value of the following statement:

$$n^2 \in \Omega(n!)$$

1. True
2. False

True/False

Determine the truth value of the following statement:

$$n^2 \in \Omega(n!)$$

1. True
2. False

True/False

Determine the truth value of the following statement:

$$(\log n)^6 \in O(n^{\frac{1}{6}})$$

1. True
2. False

True/False

Determine the truth value of the following statement:

$$(\log n)^6 \in O(n^{\frac{1}{6}})$$

1. True
2. False

True/False

Determine the truth value of the following statement:

$$(\log n)^6 \in O(n^{\frac{1}{6}})$$

1. True
2. False

General Rule: Any polynomial (n^k for some positive k) always dominates any logarithm ($(\log n)^\ell$ for some ℓ).

True/False

Determine the truth value of the following statement:

If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.

1. True
2. False

True/False

Determine the truth value of the following statement:

If $f \in O(g)$ *and* $g \in O(h)$*, then* $f \in O(h)$*.*

1. True
2. False

True/False

Determine the truth value of the following statement:

> *A* **preorder** *traversal of a binary search tree will visit the nodes in* **ascending** *order of their keys.*

1. True
2. False

True/False

Determine the truth value of the following statement:

> *A* **preorder** *traversal of a binary search tree will visit the nodes in* **ascending** *order of their keys.*

1. True
2. False

True/False

Determine the truth value of the following statement:

> *A* **preorder** *traversal of a binary search tree will visit the nodes in* **ascending** *order of their keys.*

1. True
2. False

An **inorder** traversal of a BST will visit the nodes in ascending order.

True/False

Determine the truth value of the following statement:

If $\log f(n) \in \Theta(\log g(n))$, *then* $f \in \Theta(g)$.

1. True
2. False

True/False

Determine the truth value of the following statement:

If $\log f(n) \in \Theta(\log g(n))$, *then* $f \in \Theta(g)$.

1. True
2. False

True/False

Determine the truth value of the following statement:

If $\log f(n) \in \Theta(\log g(n))$, *then* $f \in \Theta(g)$.

1. True
2. False

Counter-example: Let $f(n) = n$ and $g(n) = n^2$. Then $\log f(n) = \log n$ and $\log g(n) = \log\left(n^2\right) = 2\log n$.

True/False

Determine the truth value of the following statement:

$$n^{\log_2 5} \in O(n^2 \log n)$$

1. True
2. False

True/False

Determine the truth value of the following statement:

$$n^{\log_2 5} \in O(n^2 \log n)$$

1. True
2. False

True/False

Determine the truth value of the following statement:

$$n^{\log_2 5} \in O(n^2 \log n)$$

1. True
2. False

Because $\log_2 5 \approx 2.3 > 2$, $n^{\log_2 5}$ dominates $n^2 \log n$ because $n^{\log_2 5 - 2}$ dominates $\log n$ (any polynomial dominates any logarithm).

True/False

Determine the truth value of the following statement:

> *Let $f \in O(g)$, and let $c > 0$. If $f(n)$ and $g(n)$ are always greater than one,*
>
> $$f(n)\log_2\left(f(n)^c\right) \in O(g(n)\log_2(g(n)))$$

1. True
2. False

True/False

Determine the truth value of the following statement:

> *Let $f \in O(g)$, and let $c > 0$. If $f(n)$ and $g(n)$ are always greater than one,*
>
> $$f(n)\log_2(f(n)^c) \in O(g(n)\log_2(g(n)))$$

1. True
2. False

True/False

Determine the truth value of the following statement:

Let $f \in O(g)$, and let $c > 0$. If $f(n)$ and $g(n)$ are always greater than one,

$$f(n)\log_2\left(f(n)^c\right) \in O(g(n)\log_2\left(g(n)\right))$$

1. True
2. False

Because the constant c is in the exponent of a logarithm, it can be pulled out:

$$f(n)\log_2\left(f(n)^c\right) = cf(n)\log_2\left(f(n)\right)$$

Which will not affect the big-O relationship.

Multiple Choice

Select the strongest correct answer:

> *The* **average-case** *running time to* `insert()` *into a binary search tree with* n *nodes is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

> *The* **average-case** *running time to* `insert()` *into a binary search tree with n nodes is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

> *The* **best-case** *running time to* `insert()` *into a binary search tree with* n *nodes is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

> *The* **best-case** *running time to* `insert()` *into a binary search tree with n nodes is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

> *The* **worst-case** *running time to* `insert()` *into a binary search tree with n nodes is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

> *The* **worst-case** *running time to* `insert()` *into a binary search tree with n nodes is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

> *The amount of memory needed to store a graph with v vertices and e edges using an* **adjacency matrix** *is*

1. $O(v + e)$
2. $O(ve)$
3. $O(e^2)$
4. $O(v^2)$

Multiple Choice

Select the strongest correct answer:

> *The amount of memory needed to store a graph with v vertices and e edges using an* **adjacency matrix** *is*

1. $O(v + e)$
2. $O(ve)$
3. $O(e^2)$
4. $O(v^2)$

Multiple Choice

Select the strongest correct answer:

> *The amount of memory needed to store a graph with v vertices and e edges using an* **adjacency list** *is*

1. $O(v + e)$
2. $O(ve)$
3. $O(e^2)$
4. $O(v^2)$

Multiple Choice

Select the strongest correct answer:

> *The amount of memory needed to store a graph with v vertices and e edges using an* **adjacency list** *is*

1. $O(v + e)$
2. $O(ve)$
3. $O(e^2)$
4. $O(v^2)$

Multiple Choice

Select the strongest correct answer:

> *The* **average-case** *running time for* `put()` *into a hash table with* n *entries is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

> *The* **average-case** *running time for* `put()` *into a hash table with n entries is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

The **worst-case** *running time for* `get()` *into a hash table with n entries is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Multiple Choice

Select the strongest correct answer:

> *The* **worst-case** *running time for* `get()` *into a hash table with* n *entries is*

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$

Asymptotic Analysis

The sum of the haromic series $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \ldots$ diverges; that is:

$$\sum_{i=1}^{\infty} \frac{1}{i} = \infty$$

However, for large n, the sum of the first n terms of this series can be well approximated as

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n + \gamma$$

Where ln is the natural logarithm and γ is a constant approximately equal to 0.57721. Show (prove) the following:

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$

Asymptotic Analysis

Proposition:

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$

Proof: (Show O): By decreasing each denominator to the next power of two, we can find the upper bound:

$$\begin{aligned}
\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} &< \frac{1}{1} + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \ldots \\
&\approx \left(\sum_{i=1}^{\lfloor \log n \rfloor} 1 \right) + c' \in \Theta(\log n)
\end{aligned}$$

So $\sum_{i=1}^{n} \frac{1}{i} \in O(\log n)$.

Asymptotic Analysis

Proposition:

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$

Proof (continued):
(Show Ω): By increasing each denominator to the next power of two, we can find the lower bound:

$$\begin{aligned}
\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} \quad &> \quad \frac{1}{1}+\frac{1}{2}+\frac{1}{4}+\frac{1}{4}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8}+\dots \\
&\approx \quad 1+\left(\sum_{i=1}^{\lfloor \log n \rfloor} \frac{1}{2}\right)+c' \in \Theta(\log n)
\end{aligned}$$

So $\sum_{i=1}^{n} \frac{1}{i} \in \Omega(\log n)$, and thus $\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$.

Asymptotic Analysis

Prove the following:

For any positive real constants a, b:

$$(n + a)^b \in \Theta(n^b)$$

Asymptotic Analysis

Prove the following:

For any positive real constants a, b:

$$(n+a)^b \in \Theta(n^b)$$

Proof: (Show Ω): Take $c = 1$, $N = 1$. Then because a is positive, $n + a > n$ for any n, so

$$(n+a)^b > n^b$$

and thus $(n+a)^b \in \Omega(n^b)$.
(Show O): Take $d = 2^b$ (b is a constant, so this is valid) and $N = a$. Then for any $n > N = a$, surely $n + a < 2n$, so

$$(n+a)^b < 2^b n^b = (2n)^b$$

and thus $(n+a)^b \in O(n^b)$, and $(n+a)^b \in \Theta(n^b)$. $\square$

Graphs

Suppose you have a complete graph with n vertices (that is, between each pair of distinct nodes, there exists an edge). Suppose you run the following algorithm to find the shortest path between a vertex u and a goal node. What is the running time of this algorithm:

```
public class Graph {
    public int slow-dfs(Vertex u, Vertex goal) {
        // clearly shortest path from goal to goal is zero
        if (u.equals(goal)) return 0;
        u.visited = true; // Mark the vertex u visited
        shortestPath = infinity; // a large number
        for (each edge e adjacent to u in E) {
            v = e.destination;
            if (!v.visited) {
                testPath = e.weight + slow-dfs(v, goal);
                if (testPath < shortestPath) {
                    shortestPath = testPath;
                }
            }
        }
        u.visited = false; // unvisit u, since we want to
                           // consider all orders of vertices
        return shortestPath;
    }
}
```

Graphs

Answer: Running the algorithm will take time proportional to the number of ways we can order the nodes, so this algorithm takes $O(n!)$ time.

Graphs

Suppose we have a connected graph G with all edge weights distinct. How many minimum spanning trees exist in G?

Graphs

Suppose we have a connected graph G with all edge weights distinct. How many minimum spanning trees exist in G?

If there are two minimum spanning trees A and B of G, then

- There exists some edge $e_1 \in A$ that is not in B.
- $B \cup e_1$ has a cycle C
- There is another edge $e_2 \in C$. Without loss of generality (if not, switch A and B), the weight of e_2 is greater than that of e_1, and removing it will form a spanning tree with weight smaller than B, which is a contradiction.

Binary Search Trees: Review

A binary search tree is a tree with the following properties:

- The tree is a binary tree (each node has at most two children).
- The left subtree of any node contains only keys less than that node's key
- The right subtree of any node contains only keys greater than that node's key

Binary Search Trees: Review

A binary search tree is a tree with the following properties:

- The tree is a binary tree (each node has at most two children).
- The left subtree of any node contains only keys less than that node's key
- The right subtree of any node contains only keys greater than that node's key

To `find()` an element e:

1. Start at the root. If the tree is empty, then the key is not in the tree.
2. If the root's key is e, then return the value. Otherwise:
    1. If the root's key is greater than e, then run `find()` on its left child.
    2. Otherwise, run `find()` on its right child.

Binary Search Trees: Review

To `insert()` an element, traverse the tree like `find()` until an empty tree is reached. Insert the element into that spot.

Binary Search Trees: Review

To `insert()` an element, traverse the tree like `find()` until an empty tree is reached. Insert the element into that spot.

To `remove()` an element:

1. Search for the item using `find()`.
   1. If it has 0 children, remove the node from the tree.
   2. If it has 1 child, replace the node with its child.
   3. If it has 2 children, replace the label of the node with the label of its in-order successor and remove that node.

Binary Search Trees: Review

To `insert()` an element, traverse the tree like `find()` until an empty tree is reached. Insert the element into that spot.

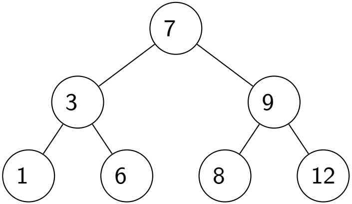
To `remove()` an element:

1. Search for the item using `find()`.
    1. If it has 0 children, remove the node from the tree.
    2. If it has 1 child, replace the node with its child.
    3. If it has 2 children, replace the label of the node with the label of its in-order successor and remove that node.

The **in-order successor** of a node is the node that is visited after the first node in an in-order traversal of the tree. In a binary search tree, the label of the in-order successor is the smallest value that is greater than the node's label. The in-order successor of a node is the bottom leftmost child in its right subtree.

Binary Search Trees

Given the following binary search tree:



Draw what it looks like after each of the following consecutive method calls:

- `insert(5)`
- `remove(3)`
- `insert(3)`
- `remove(9)`

Binary Search Trees

After `insert(5)`:

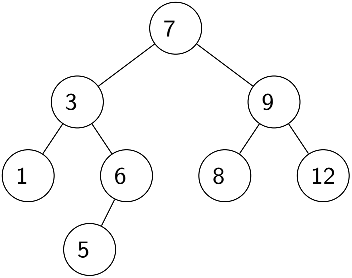
After `insert(3)`:

After `remove(9)`:

After `remove(3)`:

Binary Search Trees

After `insert(5)`:



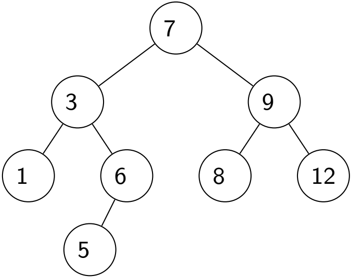
After `remove(3)`:

After `insert(3)`:

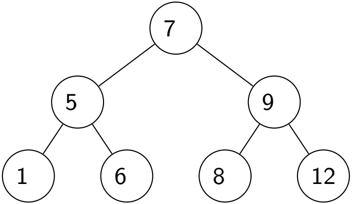
After `remove(9)`:

Binary Search Trees

After `insert(5)`:



After `remove(3)`:

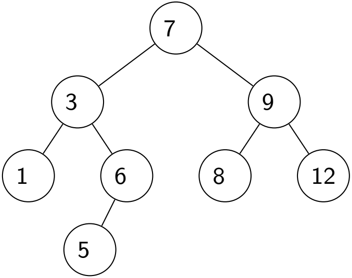


After `insert(3)`:

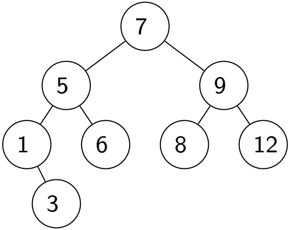
After `remove(9)`:

Binary Search Trees

After `insert(5)`:

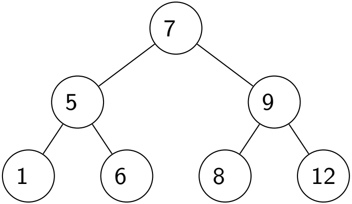


After `insert(3)`:



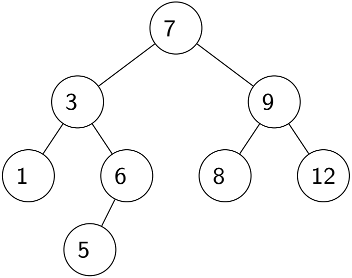
After `remove(9)`:

After `remove(3)`:

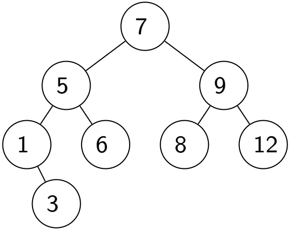


Binary Search Trees

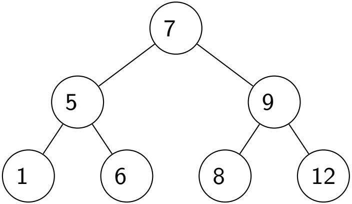
After `insert(5)`:



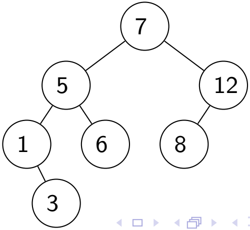
After `insert(3)`:



After `remove(3)`:



After `remove(9)`:



Binary Search Trees

If you are given the shape of a binary search tree with n nodes and a collection of n keys containing no duplicates, how many possible binary search trees can be formed?

1. 1
2. 2
3. $\log n$
4. n
5. $n!$

Binary Search Trees

If you are given the shape of a binary search tree with n nodes and a collection of n keys containing no duplicates, how many possible binary search trees can be formed?

1. 1
2. 2
3. $\log n$
4. n
5. $n!$

Binary Search Trees

If you are given the shape of a binary search tree with n nodes and a collection of n keys containing no duplicates, how many possible binary search trees can be formed?

1. 1
2. 2
3. $\log n$
4. n
5. $n!$

Given the shape of the tree, if there are ℓ nodes in the left subtree, the root must be the $\ell + 1$th smallest key. The keys can then be recursively assigned to the left and right subtrees.

Heaps: Review

A heap is a binary tree with the following properties:

- The tree is complete. That is, every level is filled except possibly the last, which is filled from left to right.
- The *heap property* or *heap invariant* holds for all nodes of the tree: If B is a descendant of A, then the key of B is greater than or equal to that of A (for a min heap).

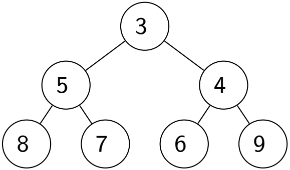
Heaps: Review

A heap is a binary tree with the following properties:

- The tree is complete. That is, every level is filled except possibly the last, which is filled from left to right.
- The *heap property* or *heap invariant* holds for all nodes of the tree: If B is a descendant of A, then the key of B is greater than or equal to that of A (for a min heap).

Heaps are usually implemented as arrays:

$\times$	3	5	4	8	7	6	9



Heaps: Review

To `insert()` an element:

1. Insert the item at the end of the array.
2. Bubble up by repeatedly swapping with parents until the heap property is satisfied.

Heaps: Review

To `insert()` an element:

1. Insert the item at the end of the array.
2. Bubble up by repeatedly swapping with parents until the heap property is satisfied.

To `removeMin()`:

1. Swap the first and last elements of the array.
2. Remove the last element and return it.
3. Bubble the root down by repeatedly comparing with both of its children and swapping until the heap property is satisfied.

Heaps

Starting with an empty **max heap**, perform the following consecutive insertions:

1. `insert(5)`
2. `insert(1)`
3. `insert(2)`
4. `insert(6)`
5. `insert(4)`
6. `insert(3)`

Draw what the heap looks like after these values are inserted.

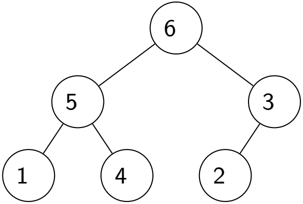
Heaps

Starting with an empty **max heap**, perform the following consecutive insertions:

1. `insert(5)`
2. `insert(1)`
3. `insert(2)`
4. `insert(6)`
5. `insert(4)`
6. `insert(3)`

Draw what the heap looks like after these values are inserted.

Answer:



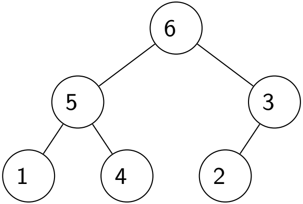
Heaps

Starting with an empty **max heap**, perform the following consecutive insertions:

1. `insert(5)`
2. `insert(1)`
3. `insert(2)`
4. `insert(6)`
5. `insert(4)`
6. `insert(3)`

Draw what the heap looks like after these values are inserted.

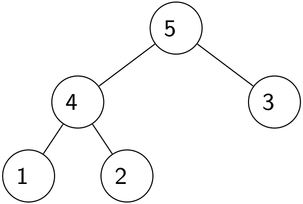
Answer:



What does the heap look like after calling `removeMax()`?

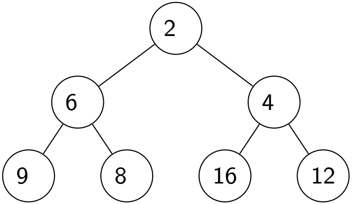
Heaps

Answer:



Heaps

Given the following **min heap**:



Draw what it looks like after each of the following consecutive method calls:

- `insert(10)`
- `removeMin()`
- `insert(3)`
- `removeMin()`

Heaps

After `insert(10)`:

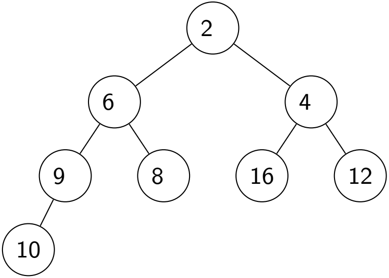
After `insert(3)`:

After `removeMin()`:

After `removeMin()`:

Heaps

After `insert(10)`:



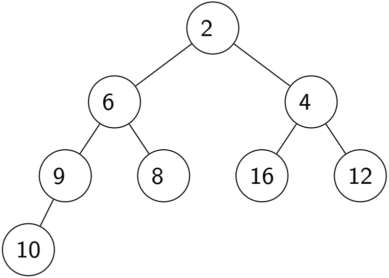
After `insert(3)`:

After `removeMin()`:

After `removeMin()`:

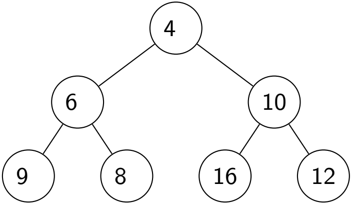
Heaps

After `insert(10)`:



After `insert(3)`:

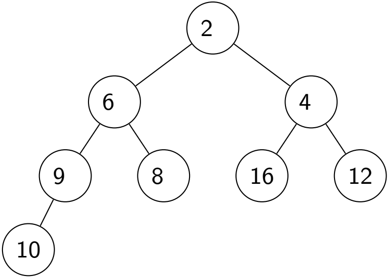
After `removeMin()`:



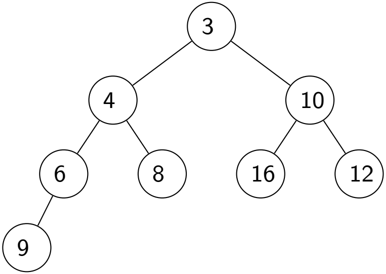
After `removeMin()`:

Heaps

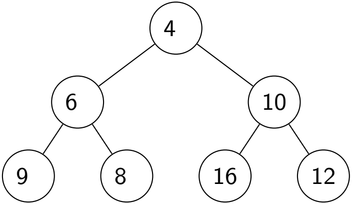
After `insert(10)`:



After `insert(3)`:



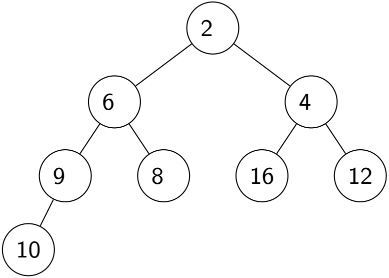
After `removeMin()`:



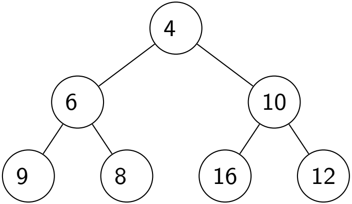
After `removeMin()`:

Heaps

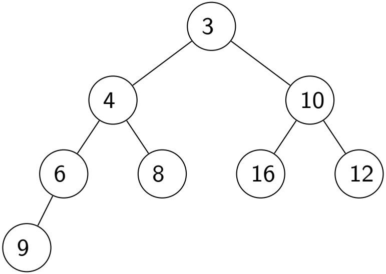
After `insert(10)`:



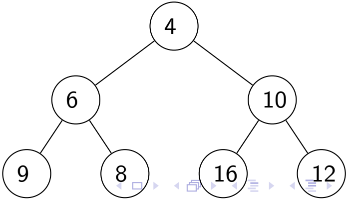
After `removeMin()`:



After `insert(3)`:



After `removeMin()`:



Heaps

Describe how you can implement a method `removeKthMin()` that, assuming there are $n > k$ nodes in the heap, removes the kth smallest item in $O(k \log n)$ time.

Heaps

Describe how you can implement a method `removeKthMin()` that, assuming there are $n > k$ nodes in the heap, removes the kth smallest item in $O(k \log n)$ time.

Answer: Call `removeMin()` k times and store the k smallest values. Insert all of them back in except for the kth smallest, and return it.
This takes $O(2k \log n) = O(k \log n)$ time.

2-3-4 Trees: Review

In a 2-3-4 tree, each node is one of the following:

- **2-node:** contains one key and has two or no children
- **3-node:** contains two keys and has three or no children
- **4-node:** contains three keys and has four or no children

2-3-4 Trees: Review

In a 2-3-4 tree, each node is one of the following:

- **2-node:** contains one key and has two or no children
- **3-node:** contains two keys and has three or no children
- **4-node:** contains three keys and has four or no children

Additionally, the following invariants are held:

- All the leaves are at the same level
- All the keys are in sorted order
    - In a 2-node, the key is greater than all the keys in the left subtree and less than all the keys in the right subtree
    - Analogous properties hold for 3-nodes and 4-nodes

2-3-4 Trees: Review

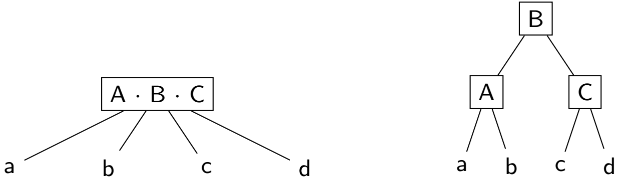
To `insert()` an element, we traverse the tree to find the correct spot to insert the key. On the way down, we fix 4-nodes. If the root is a 4-node, the tree grows by one level:

2-3-4 Trees: Review

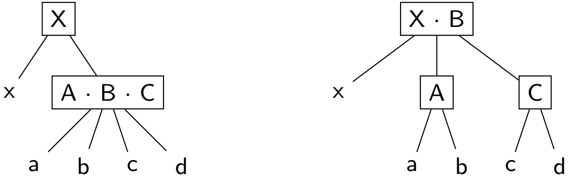
To `insert()` an element, we traverse the tree to find the correct spot to insert the key. On the way down, we fix 4-nodes. If the root is a 4-node, the tree grows by one level:

2-3-4 Trees: Review

To `insert()` an element, we traverse the tree to find the correct spot to insert the key. On the way down, we fix 4-nodes. If the root is a 4-node, the tree grows by one level:



Otherwise, the middle key is inserted into the parent (which is guaranteed not to be a 4-node).



When we finish fixing all 4-nodes, we insert it into the appropriate leaf.

2-3-4 Trees: Review

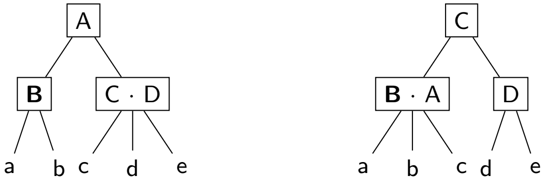
To `remove()` an element, we traverse the tree, first finding the element to remove and then the smallest key greater than it. On the way down, we fix 2-nodes. If a 2-node is reached, we first try to borrow from a sibling and perform a **rotation**:

Case 1: Rotation

2-3-4 Trees: Review

To `remove()` an element, we traverse the tree, first finding the element to remove and then the smallest key greater than it. On the way down, we fix 2-nodes. If a 2-node is reached, we first try to borrow from a sibling and perform a **rotation**:

Case 1: Rotation



2-3-4 Trees: Review

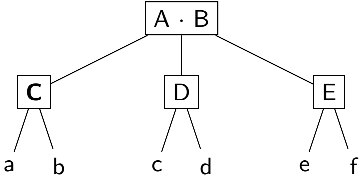
If there are no siblings that are not 2-nodes, then a key needs to be borrowed from the parent in a **fusion** operation:

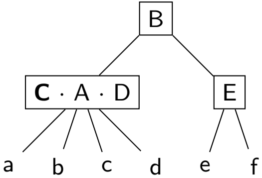
Case 2: Fusion

2-3-4 Trees: Review

If there are no siblings that are not 2-nodes, then a key needs to be borrowed from the parent in a **fusion** operation:

Case 2: Fusion





2-3-4 Trees: Review

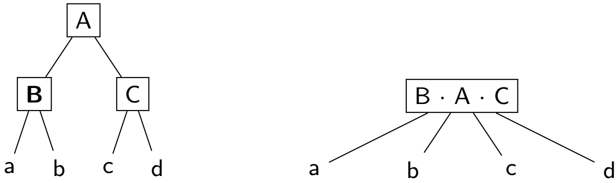
If the parent is also a 2-node (this will only happen at the root), then a special type of fusion needs to be done:

Case 3: Root Fusion

2-3-4 Trees: Review

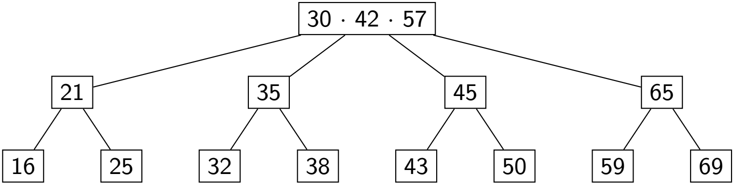
If the parent is also a 2-node (this will only happen at the root), then a special type of fusion needs to be done:

Case 3: Root Fusion



2-3-4 Trees

Given the following 2-3-4 tree:



Draw what it looks like after each of the following consecutive method calls:

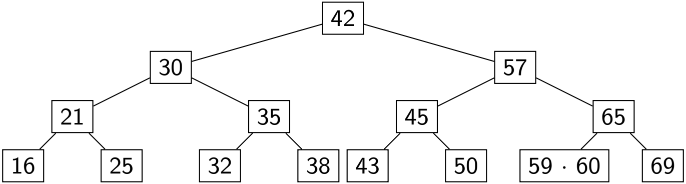
- `insert(60)`
- `insert(68)`
- `remove(59)`

2-3-4 Trees

After `insert(60)`:

2-3-4 Trees

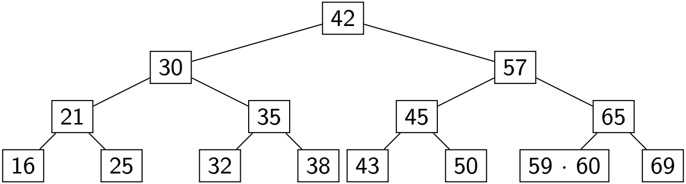
After `insert(60)`:



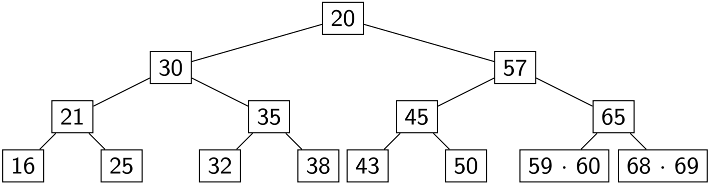
After `insert(68)`:

2-3-4 Trees

After `insert(60)`:



After `insert(68)`:

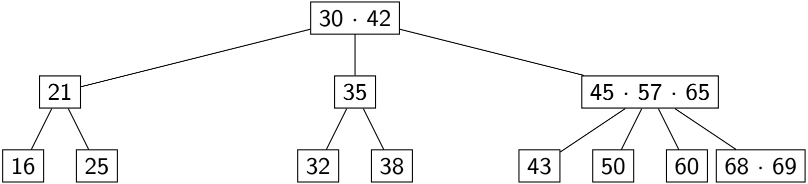


2-3-4 Trees

After `remove(59)`:

2-3-4 Trees

After `remove(59)`:



α-β Pruning

To reduce the number of nodes visited by the minimax algorithm, α-β pruning can be used:

α-β Pruning

To reduce the number of nodes visited by the minimax algorithm, α-β pruning can be used:

- α values are associated with MAX nodes. They represent the current best (highest) guaranteed score seen so far.

α-β Pruning

To reduce the number of nodes visited by the minimax algorithm, α-β pruning can be used:

- α values are associated with MAX nodes. They represent the current best (highest) guaranteed score seen so far.
- β values are associated with MIN nodes. They represent the current best (lowest) guaranteed score seen so far.

α-β Pruning

To reduce the number of nodes visited by the minimax algorithm, α-β pruning can be used:

- α values are associated with MAX nodes. They represent the current best (highest) guaranteed score seen so far.
- β values are associated with MIN nodes. They represent the current best (lowest) guaranteed score seen so far.

When we start α-β pruning, α values start at $-\infty$ and β values start at ∞. We prune on two cases:

α-β Pruning

To reduce the number of nodes visited by the minimax algorithm, α-β pruning can be used:

- α values are associated with MAX nodes. They represent the current best (highest) guaranteed score seen so far.
- β values are associated with MIN nodes. They represent the current best (lowest) guaranteed score seen so far.

When we start α-β pruning, α values start at $-\infty$ and β values start at ∞. We prune on two cases:

- Search can stop below any MIN node whose β value is less than or equal to the α value of any of its MAX ancestors.

α-β Pruning

To reduce the number of nodes visited by the minimax algorithm, α-β pruning can be used:

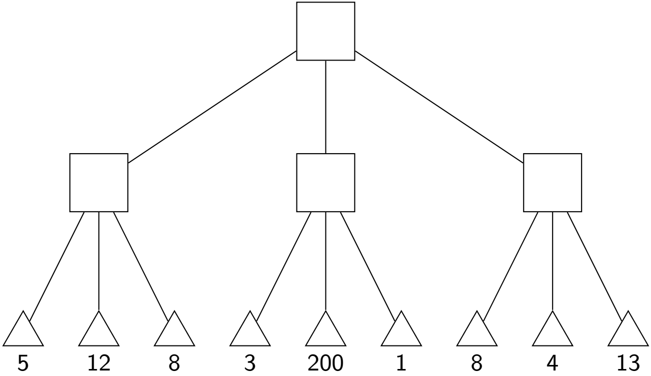
- α values are associated with MAX nodes. They represent the current best (highest) guaranteed score seen so far.
- β values are associated with MIN nodes. They represent the current best (lowest) guaranteed score seen so far.

When we start α-β pruning, α values start at $-\infty$ and β values start at ∞. We prune on two cases:

- Search can stop below any MIN node whose β value is less than or equal to the α value of any of its MAX ancestors.
- Search can stop below any MAX node whose α value is greater than or equal to the β value of any of its MIN ancestors.

α-β Pruning

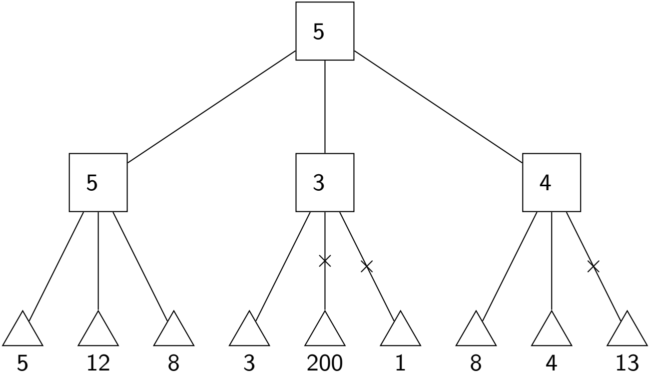
Consider the following tree below:



Determine which nodes will be pruned using α-β pruning and the final minimax value of the root. Assume the first player is MAX.

α-β Pruning

Solution:



Hash Tables

```
put(key, value):
```

Hash Tables

`put(key, value)`:

1. The key (must be an object!) has its `hashCode()` method called by the hash table

Hash Tables

`put(key, value)`:

1. The key (must be an object!) has its `hashCode()` method called by the hash table
2. The key is compressed (usually a modulo function) into the range of the number of buckets

Hash Tables

`put(key, value)`:

1. The key (must be an object!) has its `hashCode()` method called by the hash table
2. The key is compressed (usually a modulo function) into the range of the number of buckets
3. The key and value are stored into the corresponding bucket

`get(key)`:

Hash Tables

`put(key, value)`:

1. The key (must be an object!) has its `hashCode()` method called by the hash table
2. The key is compressed (usually a modulo function) into the range of the number of buckets
3. The key and value are stored into the corresponding bucket

`get(key)`:

1. The key's `hashCode()` is compressed to find the appropriate bucket

Hash Tables

`put(key, value)`:

1. The key (must be an object!) has its `hashCode()` method called by the hash table
2. The key is compressed (usually a modulo function) into the range of the number of buckets
3. The key and value are stored into the corresponding bucket

`get(key)`:

1. The key's `hashCode()` is compressed to find the appropriate bucket
2. The contents of the bucket are searched for the key

Hash Tables

`put(key, value)`:

1. The key (must be an object!) has its `hashCode()` method called by the hash table
2. The key is compressed (usually a modulo function) into the range of the number of buckets
3. The key and value are stored into the corresponding bucket

`get(key)`:

1. The key's `hashCode()` is compressed to find the appropriate bucket
2. The contents of the bucket are searched for the key

Common implementations:

Hash Tables

`put(key, value)`:

1. The key (must be an object!) has its `hashCode()` method called by the hash table
2. The key is compressed (usually a modulo function) into the range of the number of buckets
3. The key and value are stored into the corresponding bucket

`get(key)`:

1. The key's `hashCode()` is compressed to find the appropriate bucket
2. The contents of the bucket are searched for the key

Common implementations:

1. Buckets represented as an array

Hash Tables

`put(key, value)`:

1. The key (must be an object!) has its `hashCode()` method called by the hash table
2. The key is compressed (usually a modulo function) into the range of the number of buckets
3. The key and value are stored into the corresponding bucket

`get(key)`:

1. The key's `hashCode()` is compressed to find the appropriate bucket
2. The contents of the bucket are searched for the key

Common implementations:

1. Buckets represented as an array
2. Items in buckets can be put in linked lists

Analysis

What's the running time for `put()`?

Analysis

What's the running time for `put()`?

- Running time: $O(1)$ (constant time – why?)

What's the running time for `get()`?

Analysis

What's the running time for `put()`?

- Running time: $O(1)$ (constant time – why?)

What's the running time for `get()`?

- Let b be the number of buckets and n be the number of key-value pairs in the `hash table`

Analysis

What's the running time for `put()`?

- Running time: $O(1)$ (constant time – why?)

What's the running time for `get()`?

- Let b be the number of buckets and n be the number of key-value pairs in the `hash table`
- Finding the appropriate bucket is a constant-time operation (why?)

Analysis

What's the running time for `put()`?

- Running time: $O(1)$ (constant time – why?)

What's the running time for `get()`?

- Let b be the number of buckets and n be the number of key-value pairs in the `hash table`
- Finding the appropriate bucket is a constant-time operation (why?)
- On average, there are n/b elements in a bucket

Analysis

What's the running time for `put()`?

- Running time: $O(1)$ (constant time – why?)

What's the running time for `get()`?

- Let b be the number of buckets and n be the number of key-value pairs in the `hash table`
- Finding the appropriate bucket is a constant-time operation (why?)
- On average, there are n/b elements in a bucket
- Finding the key in a bucket is proportional to the number of items in the bucket

Analysis

What's the running time for `put()`?

- Running time: $O(1)$ (constant time – why?)

What's the running time for `get()`?

- Let b be the number of buckets and n be the number of key-value pairs in the `hash table`
- Finding the appropriate bucket is a constant-time operation (why?)
- On average, there are n/b elements in a bucket
- Finding the key in a bucket is proportional to the number of items in the bucket
- Running time: $O(n/b)$

Analysis

What happens if b is constant?

- We end up with a linear-time `get()`, which is no better than storing key-value pairs in an `ArrayList`!

Analysis

What happens if b is constant?

- We end up with a linear-time `get()`, which is no better than storing key-value pairs in an `ArrayList`!

Solution: Keep b proportional to n

Analysis

What happens if b is constant?

- We end up with a linear-time `get()`, which is no better than storing key-value pairs in an `ArrayList`!

Solution: Keep b proportional to n

- Let k be the proportionality constant – define $k = n/b =$ average number of key-value pairs in a bucket

Analysis

What happens if b is constant?

- We end up with a linear-time `get()`, which is no better than storing key-value pairs in an `ArrayList`!

Solution: Keep b proportional to n

- Let k be the proportionality constant – define $k = n/b =$ average number of key-value pairs in a bucket
- Once we have more than $b * k$ key-value pairs, we double the size of the array and re-hash all of our key-value pairs

Analysis

What happens if b is constant?

- We end up with a linear-time `get()`, which is no better than storing key-value pairs in an `ArrayList`!

Solution: Keep b proportional to n

- Let k be the proportionality constant – define $k = n/b =$ average number of key-value pairs in a bucket
- Once we have more than $b * k$ key-value pairs, we double the size of the array and re-hash all of our key-value pairs
- How long does this take?

Analysis

What happens if b is constant?

- We end up with a linear-time `get()`, which is no better than storing key-value pairs in an `ArrayList`!

Solution: Keep b proportional to n

- Let k be the proportionality constant – define $k = n/b =$ average number of key-value pairs in a bucket
- Once we have more than $b * k$ key-value pairs, we double the size of the array and re-hash all of our key-value pairs
- How long does this take?
  - Creating a new array takes $O(n)$ time

Analysis

What happens if b is constant?

- We end up with a linear-time `get()`, which is no better than storing key-value pairs in an `ArrayList`!

Solution: Keep b proportional to n

- Let k be the proportionality constant – define $k = n/b =$ average number of key-value pairs in a bucket
- Once we have more than $b * k$ key-value pairs, we double the size of the array and re-hash all of our key-value pairs
- How long does this take?
  - Creating a new array takes $O(n)$ time
  - For each key-value pair, `put(key, value)` takes constant time – there are $b * k$ pairs, so it takes $O(b * k) = O(n)$ time

Hash Tables

Analysis

How long will `put()` take?

Hash Tables

Analysis

How long will `put()` take?

- Take average running time – divide the running time to insert n key-value pairs by n

Hash Tables

Analysis

How long will `put()` take?

- Take average running time – divide the running time to insert n key-value pairs by n
  - We need to double the array $\log n$ times, which will take

$$\cdots + n/8 + n/4 + n/2 + n = 2n = O(n)$$

Hash Tables

Analysis

How long will `put()` take?

- Take average running time – divide the running time to insert n key-value pairs by n
    - We need to double the array $\log n$ times, which will take

$$\cdots + n/8 + n/4 + n/2 + n = 2n = O(n)$$

    - Other than doubling the array, insertion takes constant time for each key-value pair, so an additional $O(n)$ time required

Hash Tables

Analysis

How long will `put()` take?

- Take average running time – divide the running time to insert n key-value pairs by n
  - We need to double the array $\log n$ times, which will take

$$\cdots + n/8 + n/4 + n/2 + n = 2n = O(n)$$

  - Other than doubling the array, insertion takes constant time for each key-value pair, so an additional $O(n)$ time required
  - Total time: $O(n + n) = O(n)$ time, so average (amortized) running time is $O(n/n) = O(1)$ time – constant time!

How long will `get()` take?

Hash Tables

Analysis

How long will `put()` take?

- Take average running time – divide the running time to insert n key-value pairs by n
  - We need to double the array $\log n$ times, which will take

$$\cdots + n/8 + n/4 + n/2 + n = 2n = O(n)$$

  - Other than doubling the array, insertion takes constant time for each key-value pair, so an additional $O(n)$ time required
  - Total time: $O(n + n) = O(n)$ time, so average (amortized) running time is $O(n/n) = O(1)$ time – constant time!

How long will `get()` take?

- The average bucket size is no greater than k (a constant)

Hash Tables

Analysis

How long will `put()` take?

- Take average running time – divide the running time to insert n key-value pairs by n
  - We need to double the array $\log n$ times, which will take
$$\cdots + n/8 + n/4 + n/2 + n = 2n = O(n)$$
  - Other than doubling the array, insertion takes constant time for each key-value pair, so an additional $O(n)$ time required
  - Total time: $O(n + n) = O(n)$ time, so average (amortized) running time is $O(n/n) = O(1)$ time – constant time!

How long will `get()` take?

- The average bucket size is no greater than k (a constant)
- Time to search a bucket is proportional to k

Hash Tables

Analysis

How long will `put()` take?

- Take average running time – divide the running time to insert n key-value pairs by n
  - We need to double the array $\log n$ times, which will take

$$\cdots + n/8 + n/4 + n/2 + n = 2n = O(n)$$

  - Other than doubling the array, insertion takes constant time for each key-value pair, so an additional $O(n)$ time required
  - Total time: $O(n+n) = O(n)$ time, so average (amortized) running time is $O(n/n) = O(1)$ time – constant time!

How long will `get()` take?

- The average bucket size is no greater than k (a constant)
- Time to search a bucket is proportional to k
- Running time: $O(k) = O(1)$

Hash Codes

Review

What makes a hash code good?

Hash Codes

Review

What makes a hash code good?

- If two numbers are `.equals()`, then their hash codes **must** be the same!

Hash Codes

Review

What makes a hash code good?

- If two numbers are `.equals()`, then their hash codes **must** be the same!
- Avoid collisions

Hash Codes

Review

What makes a hash code good?

- If two numbers are `.equals()`, then their hash codes **must** be the same!
- Avoid collisions
  - Collisions – two objects have the same hash code

Hash Codes

Review

What makes a hash code good?

- If two numbers are `.equals()`, then their hash codes **must** be the same!
- Avoid collisions
  - Collisions – two objects have the same hash code
  - Too many collisions for one bucket will slow down `get()`

Hash Codes

Review

What makes a hash code good?

- If two numbers are `.equals()`, then their hash codes **must** be the same!
- Avoid collisions
  - Collisions – two objects have the same hash code
  - Too many collisions for one bucket will slow down `get()`
- Should be able to compute hash codes quickly