CSCI 241

Lecture 13:
One last bit of AVL
Priority Queue
Heaps
Announcements

• Welcome back! It’s been a while.
• A2 is due Tuesday night:
  • Look through the supplemental AVL slides!
  • Visit me in office hours!
  • Email me if you’re stuck over the weekend!
  • I’ll post an FAQ if I get many duplicate questions.
• Midterm exam is next Friday.
  • Covers material through today.
  • Study guide will be available by Monday.
  • One double-sided sheet of hand-written notes is allowed.
  • Quizzes, ABCDs, etc. are the most efficient study tool.
Goals

• Understand some efficiency gotchas:
  • A1 - copying the array in merge
  • A2 - computing height in rebalance

• Understand the purpose and interface of the Priority Queue ADT.

• Know the definition and properties of a heap.

• Know how heaps are stored in practice.

• Know how to implement add, peek, and poll heap operations.
merge(A, start, mid, end):
  B = deep copy of A
  i = start
  j = mid
  k = 0
  while i < mid and j < end:
    if B[i] < B[j]:
      A[k] = B[i]
      i++
    else:
      A[k] = B[j]
      j++
      k++
  while i < mid:
    A[k] = B[i]
    i++, k++
  while j < end:
    A[k] = B[j]
    j++, k++
A1 Efficiency Gotcha

- In merge(), copying the entire array A makes it $O(A.length)$

- Our runtime analysis relied on merge being $O(A[end-start])$

- Copying all of A instead of the relevant range of A makes mergesort $O(n^2)$!
AVL Insertion

/* insert a node with value v into the
 * tree rooted at n. pre: n is not null. */
insert(Node n, int v):
    if n.value == v: return // (duplicate)
    if v < n.value:
        if n has left:
            insert(n.left, v)
        else:
            // attach new node w/ value v to n.left
    else: // v > n.value
        if n has right:
            insert(n.right, v)
        else:
            // attach new node w/ value v to n.right
rebalance(n);
AVL Rebalance

Read the supplemental AVL slides. Do the exercises. Convince yourself that the code for rebalance does the right thing.

An insertion that unbalances $n$ could have gone one of four places.  

Case 1  
Case 2  
Case 3  
Case 4  

Symmetric
AVL Rebalance

Before an insertion that unbalances n, the tree must look like one of these:

An insertion that unbalances n could have gone one of four places.

Case 1
Case 2
Case 3
Case 4
void rebalance(n):
    if bal(n) < -1:
        if bal(n.left) < 0
            // case 1:
            // rightRot(n)
        else
            // case 2:
            // leftRot(n.L);
            // rightRot(n)
    else if bal(n) > 1:
        if bal(n.right) < 0:
            // case 3:
            // rightRot(n.R);
            // leftRot(n)
        else:
            // case 4:
            // leftRot(n)
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case 4:
      // leftRot(n)
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case 4:
      // leftRot(n)

Cases 3 and 4 are symmetric with 2 and 1.
void rebalance(n):
    if bal(n) < -1:
        if bal(n.left) < 0
            // case 1:
            // rightRot(n)
        else:
            // case 2:
            // leftRot(n.L);
            // rightRot(n)
    else:
        // case 2:
        // leftRot(n.L);
        // rightRot(n)
    else if bal(n) > 1:
        if bal(n.right) < 0:
            // case 3:
            // rightRot(n.R);
            // leftRot(n)
        else:
            // case 4:
            // leftRot(n)

Cases 3 and 4 are symmetric with 2 and 1.
Maintaining Height

```c
void rebalance(n):
    if bal(n) < -1:
        if bal(n.left) < 0
            // case 1:
            // rightRot(n)
        else:
            // case 2:
            // leftRot(n.L);
            // rightRot(n)
    else if bal(n) > 1:
        if bal(n.right) < 0:
            // case 3:
            // rightRot(n.R);
            // leftRot(n)
        else:
            // case 4:
            // leftRot(n)
```

- `bal` needs to know children’s heights.
- Computing height by recursively walking the tree (as in lab3) is \(O(n)\)!
- Doing this every time we compute `bal` would ruin our \(O(\log n)\) runtime of AVL insertion!
- Each node needs to keep track of its height.
Maintaining Height

• Each node needs to keep track of its height.

  • Key idea: if my childrens’ heights are correct, my height can be calculated in constant time:

  \[ n\.height = 1 + \max(n\.left\.height, n\.right\.height) \]

• When can a node’s height change?

  • After a rotation
  
  • After an insertion
Maintaining Height

• Each node needs to keep track of its height.

• Key idea: if my childrens’ heights are correct, my height can be calculated in constant time:
  \[ n.height = 1 + \max(n.left.height, n.right.height) \]

• When can a node’s height change?
  • After a rotation
  • After an insertion
Height after rotations

- Heights of child subtrees (alpha, beta, gamma) can't change!

- Heights of x and y change, but can be calculated directly from (already-correct) heights of children.
Maintaining Height

• Each node needs to keep track of its height.

  • Key idea: if my childrens’ heights are correct, my height can be calculated in constant time:

    • $n\cdot\text{height} = 1 + \max(n\cdot\text{left}\cdot\text{height}, n\cdot\text{right}\cdot\text{height})$

• When can a node’s height change?

  • After a rotation

  • After an insertion
Height after insertion

2 0
10 2
15 1
16 0
18 0
• All nodes on the path to root need to have height updated.

• We’re calling rebalance on exactly those nodes!

• Each node’s height update can safely be computed from the child heights.
• All nodes on the path to root need to have height updated.

• We’re calling rebalance on exactly those nodes!

• Each node’s height update can safely be computed from the child heights.
• All nodes on the path to root need to have height updated.

• We’re calling rebalance on exactly those nodes!

• Each node’s height update can safely be computed from the child heights.
• All nodes on the path to root need to have height updated.

• We’re calling rebalance on exactly those nodes!

• The calls happen bottom-up, so each child heights have been updated already.
Height of AVL Trees

• As usual, runtime of search, insert, and remove are all $O(\text{height})$.

• A rotation is $O(1)$, so even if we have to rebalance every node on the path to the root, it’s still only $h \times O(1)$ rebalances.
Height Updates during Insertion
(another visualization for your reference)

```python
insert(Node n, int v):
    // ...(other case, irrelevant here)
    else: // v > n.value
        if n has right:
            insert(n.right, v)
        else:
            // attach new node w/ value
            //   v to n.right
            rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
=>insert(f, 16)
=>attach new node
    rebalance(f)
    rebalance(c)
    rebalance(a)
```
AVL Insertion

```
insert(Node n, int v):
    // ...(other case, irrelevant here)
else: // v > n.value
    if n has right:
        insert(n.right, v)
    else:
        // attach new node w/ value
        //  v to n.right
    rebalance(n);
```

insert(a, 16)
=>insert(c, 16)
=>insert(f, 16)
=>attach new node
    rebalance(f)
    rebalance(c)
    rebalance(a)
AVL Insertion

insert(Node n, int v):
    // ...(other case, irrelevant here)
    else: // v > n.value
        if n has right:
            insert(n.right, v)
        else:
            // attach new node w/ value
            // v to n.right
            rebalance(n);

insert(a, 16)
=>insert(c, 16)
=>insert(f, 16)
=>attach new node
    rebalance(f)
    rebalance(c)
    rebalance(a)

update f’s height
AVL Insertion

\[\text{insert}(\text{Node } n, \text{ int } v):\]
\[\quad /\ldots(\text{other case, irrelevant here})\]
\[\quad \text{else: } \quad /\ v > n.\text{value}\]
\[\quad \quad \text{if } n \text{ has right:}\]
\[\quad \quad \quad \text{insert}(n.\text{right}, v)\]
\[\quad \quad \text{else:}\]
\[\quad \quad \quad \quad /\ attach \ new \ node \ w/ \ value\]
\[\quad \quad \quad \quad \quad /\ v \ to \ n.\text{right}\]
\[\quad \quad \text{rebalance}(n);\]

\[\text{insert}(a, 16)\]
\[\Rightarrow \text{insert}(c, 16)\]
\[\Rightarrow \text{insert}(f, 16)\]
\[\Rightarrow \text{attach new node}\]
\[\text{rebalance}(f)\]
\[\text{rebalance}(c)\]
\[\text{rebalance}(a)\]
AVL Insertion

```python
insert(Node n, int v):
    // ...(other case, irrelevant here)
    else: // v > n.value
        if n has right:
            insert(n.right, v)
        else:
            // attach new node w/ value
            //   v to n.right
            rebalance(n);
```

Insertions:
- `insert(a, 16)`
  => `insert(c, 16)`
  => `insert(f, 16)`
  => `attach new node
      // to n.right`
  => `rebalance(f)`
  => `rebalance(c)`
  => `rebalance(a)`

Update heights:
- `update f's height`
- `update c's height`
- `(rotation happens)`
AVL Insertion

```
insert(Node n, int v):
    //...(other case, irrelevant here)
else: // v > n.value
    if n has right:
        insert(n.right, v)
    else:
        // attach new node w/ value
        // v to n.right
        rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
=>insert(f, 16)
=>attach new node
    rebalance(f)
rebalance(c)
rebalance(a)
```

```
update f’s height
update c’s height
update a’s height
```
Height of AVL Trees

• As usual, runtime of search, insert, and remove are all $O(\text{height})$

• What is the worst-case height of an AVL tree with $n$ nodes?
  • Exact proof is CSCI 405 material; uses the fibonacci sequence(!)
  • Spoiler: the answer is $O(\log n)$

• Intuition: To add to root’s height, you have to add to height of every subtree in one of root’s subtrees.
Removing from AVL Tree

• Much like insertion: remove as usual, rebalance as necessary at each level up to the root.

• Whereas insertion only ever requires only one rebalance, deletion can require many
  • but still no more than the tree’s height.
Priority Queues
Queue vs Priority Queue

add (enqueue): inserts an item into the queue
remove (dequeue): removes the first item to be inserted (FIFO)

add (enqueue): inserts an item into the queue
remove (poll): remove the highest-priority item from the queue
Uses for Priority Queues

- Computer Graphics: mesh simplification
- Graph algorithms: shortest paths, spanning trees
- Statistics: maintain largest M values in a sequence
- Graphics and simulation: "next time of contact" for colliding bodies
- AI Path Planning: A* search (e.g., Map directions)
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling
Priority Queues

Like a Queue, but:

• Each item in the queue has an associated priority which is some type that implements Comparable
• `remove()` returns item with the “highest priority”
  • or, the element with the “smallest” associated priority value
• Ties are broken arbitrarily
interface PriorityQueue<E> {
    boolean add(E e); // insert e
    E peek(); // return min element
    E poll(); // remove/return min element
    void clear();
    boolean contains(E e);
    boolean remove(E e);
    int size();
    Iterator<E> iterator();
}
Priority Queue: LinkedList implementation

An unordered list:

• **add()** - new element at front of list - $O(1)$
• **poll()** - requires searching the list - $O(n)$
• **peek()** - requires searching the list - $O(n)$

An ordered list:

• **add()** - requires searching the list - $O(n)$
• **poll()** - min element is kept at front - $O(1)$
• **peek()** - min element is kept at front - $O(1)$
Question to ponder:

What would be the runtime of add, peek, and poll if you implement a Priority Queue using a BST?

What about an AVL tree?
Priority Queue: heap implementation

• A heap is a **concrete** data structure that can be used to **implement** a Priority Queue

• Better runtime complexity than either list implementation:
  • `peek()` is $O(1)$
  • `poll()` is $O(\log n)$
  • `add()` is $O(\log n)$

• Not to be confused with *heap memory*, where the Java virtual machine allocates space for objects – different usage of the word heap.
A heap is a special binary tree with two additional properties.
A heap is a special binary tree.

1. **Heap Order Invariant:**
   Each element $\geq$ its parent.
A heap is a special binary tree.

2. **Complete**: no holes!
- All levels except the last are full.
- Nodes in last level are as far left as possible.

```
    4
   /|
  6 14
 /|
21 8 35
/|
22 38 55 10 19 35
```

← as far left as possible
Heap it real

Which of these are valid heaps?
Heap it real.

(A) 5
   / \  
  12  15
 / \  
13 11

(B) 5
   / \  
  12  15
 / \  
13 14

(C) 5
   / \  
  12  15
 / \  
13 16

(D) -5
   /   \  
  12   15
 /     
18 13 15

Which of these are valid heaps?
Heap operations

interface PriorityQueue<E> {
    boolean add(E e); // insert e
    E peek(); // return min element
    E poll(); // remove/return min element
    void clear();
    boolean contains(E e);
    boolean remove(E e);
    int size();
    Iterator<E> iterator();
}
```java
boolean add(E e);
```

**Algorithm:**
- Add e in the wrong place
- While e is in the wrong place
  - move e towards the right place
boolean add(E e);
boolean add(E e);
boolean add(E e);
boolean add(E e);
boolean add(E e);
boolean add(E e);

Algorithm:
• Add e in the wrong place (the leftmost empty leaf)
• While e is in the wrong place (it is less than its parent)
  • move e towards the right place (swap with parent)

The heap invariant is maintained!
What’s the runtime?

- $O(\text{number of swap/bubble operations})$
  
  $= O(\text{height of tree})$

- Complete => balanced
  
  => height is $O(\log n)$

- Maximum number of swaps is $O(\log n)$
add(e)

**Algorithm:**
- Add e in the wrong place (the leftmost empty leaf)
- While e is in the wrong place (it is less than its parent)
  - move e towards the right place (swap with parent)

The heap invariant is maintained!
public class HeapNode {
    private int value;
    private HeapNode left;
    private HeapNode right;
    ...
}

public class Heap {
    HeapNode root;
    ...
}
public class Heap
{
    private int value;
    private Heap left;
    private Heap right;
    ...
A heap is a special binary tree.

2. **Complete**: no holes!
Numbering Nodes

Level-order traversal:

2. Complete: no holes!
node $k$’s parent is
node $k$’s children are nodes and
node \( k \)'s parent is \( \frac{(k - 1)}{2} \)

node \( k \)'s children are nodes and
Numbering Nodes

node $k$’s parent is $(k - 1)/2$
node $k$’s children are nodes $2k + 1$ and $2k + 2$
Implementing Heaps

```java
public class Heap<E> {
    private E[] heap;
    private int size;
    ...
}
```
Implicit Tree Structure

2. Complete: **no holes!**