Algorithms and Data Structures for Data Science

Binary Search Tree

CS 277
Brad Solomon

March 6, 2024
Reminder: mp_automata due Friday

93% credit late day extension through Saturday

Additional extensions by request
Reminder: Spring Break next week

Lab on Friday will still happen, will be due after spring break

No office hours during spring break
Exam 2: 3/19 - 3/21

Yes its right after spring break. Sorry!

Covered material described on website

One coding question — likely similar to mp_automata

Practice exam (hopefully) later this week
Learning Objectives

Finish implementation of BST ADT

Introduce the Huffman Tree

Practice recursion in the context of trees
A BST is a binary tree $T = \text{treeNode}(val, T_L, T_r)$ such that:

$\forall n \in T_L, n.val < T.val$

Left is smaller

$\forall n \in T_R, n.val > T.val$

Right is larger

```
class bstNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```
Binary **Search** Tree ADT — *what changed?*

**Constructor:** Build a new (empty) tree

**Insert:** Find the correct insert location based on BST structure

**Remove:** Find the node being removed and… ???

**Traverse:** Visit every node in tree (all objects)

**Search:** Find a specific node in the tree using the ‘key’ value
BST Remove

0 child case

1) Find node to be removed

2) Set parent.child = None

- remove @ 38
  - remove @ 51
  - remove @ 10
  - return None

node.left = (remove @ 40)
node@ 51.left = None

remove(40)
BST Remove

1 child case

Linked List!

Set parent.child = p.child.child

remove @ 38
  \rightarrow remove @ 13
    \rightarrow remove @ 25

return 25\rightarrow right

node.right = remove @ 25
BST Remove

2 child case
1) Find Node being removed
2) Swap w/ IOP or IOS

GOOD: IOP / IOS is swap key, value

BAD:

Good

10
12
25
37

In-order predecessor
In-order successor

remove(13)
def remove(self, key):
    self.root = self.remove_helper(self.root, key)

def remove_helper(self, node, key):
    1) Find Node
    2) Find LCP/sup
    3) Swap Values
    4) Remove Node
    5) Return Node

Removal process:
- Find the node with the specified key.
- If the node has no children, remove it.
- If the node has one child, replace the node with its child.
- If the node has two children, replace the node with its LCP/sup (largest value in the left subtree).
- Recursively remove the LCP/sup from its subtree.

BST Remove
What will the tree structure look like if we remove node 16 using IOS?

1) What is $\text{I}os(16) - 18$

2) Swap with 18

3) Remove node 16
### BST Analysis – Running Time

<table>
<thead>
<tr>
<th>Operation</th>
<th>BST Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>find</strong></td>
<td>(O(n))</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>(O(n))</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>(O(n))</td>
</tr>
<tr>
<td><strong>traverse</strong></td>
<td>(O(n))</td>
</tr>
</tbody>
</table>
Limiting the height of a tree

- The height of a tree can be limited by bounding it to \( \log n \) levels.
- The height of a tree is \( \log \) of the number of nodes.

Diagram:
- Tree with nodes 1, 5, 7, 9, 6, 9, 7, 5, 1.
Height-Balanced Tree

What tree is better?

How would you describe this mathematically?
Height-Balanced Tree

What tree is better?

Height balance: \( b = \text{height}(T_R) - \text{height}(T_L) \)

A tree is “balanced” if: all nodes have \( b < 2 \)
Option A: Correcting bad insert order

The height of a BST depends on the order in which the data was inserted

**Insert Order:** [1, 3, 2, 4, 5, 6, 7]

**Insert Order:** [4, 2, 3, 6, 7, 1, 5]
AVL-Tree: A self-balancing binary search tree

Rather than fixing an insertion order, just correct the tree as needed!
We will return to this topic... after spring break!
Optimal Storage Costs

Achieving an optimal storage cost for a dataset is often important.

Let's use strings as an accessible example!

What is the minimum bits needed to encode the message:

'feed me more food'

<table>
<thead>
<tr>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>000</td>
</tr>
<tr>
<td>e</td>
<td>001</td>
</tr>
<tr>
<td>d</td>
<td>010</td>
</tr>
<tr>
<td>m</td>
<td>100</td>
</tr>
<tr>
<td>r</td>
<td>011</td>
</tr>
<tr>
<td>o</td>
<td>101</td>
</tr>
<tr>
<td>' '</td>
<td>110</td>
</tr>
</tbody>
</table>

$2^3 = 8$

7 characters = 2.45

11 - 8th char
Optimal Storage Costs

Using three bits per character, we have 51 bits total. But can we do better?

‘feed me more food’

If we think about our input as a sorted list of frequencies, yes!

r: 1  d: 2  f: 2  m: 2  o: 3  'SPACE': 3  e: 4

\[ \text{Count \# of each character} \]

\[ \text{Few bits \iff high freq} \]

\[ \text{More bits \iff low freq} \]
Using binary trees for string encoding

Let's define a tree with the following:

The keys are individual characters

The values are the frequencies of those characters

```python
class treeNode:
    def __init__(self, key, val, left=None, right=None):
        self.key = key
        self.val = val
        self.left = left
        self.right = right
```

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>7</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
</tr>
</tbody>
</table>

Leaves are single characters.
Binary Tree encoding

Given the following two trees, how might we define an encoding?
Binary Tree encoding

How did we produce this encoding?

<table>
<thead>
<tr>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>010</td>
</tr>
<tr>
<td>D</td>
<td>011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>11</td>
</tr>
</tbody>
</table>
Binary Tree encoding

The path from root to leaf defines our encoding, but which tree is best?

Going left = 0

Going right = 1

<table>
<thead>
<tr>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>010</td>
</tr>
<tr>
<td>D</td>
<td>011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
</tr>
<tr>
<td>D</td>
<td>11</td>
</tr>
</tbody>
</table>
Binary Tree encoding

If my frequencies are \{A : 7 | B : 5 | C : 2 | D : 4 \}, which tree was better?

<table>
<thead>
<tr>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>010</td>
</tr>
<tr>
<td>D</td>
<td>011</td>
</tr>
</tbody>
</table>

Build a tree based on frequencies

Better
Building the Huffman Tree

The **Huffman Tree** is the tree with the optimal total path length for a given set of characters and their frequencies.

**Step 1: Calculate the frequency of every character in text** and order by increasing frequency. Store in a queue (a sorted list).

**Input:** 'feed me more food'

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>m</td>
<td>2</td>
</tr>
<tr>
<td>o</td>
<td>3</td>
</tr>
<tr>
<td>'SPACE'</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
</tr>
</tbody>
</table>

Only remove Smallest

Only add larger
Building the Huffman Tree

**Step 2: Build a tree from the bottom up.** Start by taking the two least frequent characters and merge them (create a parent node). Store the merged characters in a new queue.

**Input:**

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>2</td>
</tr>
<tr>
<td>m</td>
<td>2</td>
</tr>
<tr>
<td>o</td>
<td>3</td>
</tr>
<tr>
<td>SPACE</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
</tr>
</tbody>
</table>

Muse:

1. **Concatenate strings / characters**
   
   cf d = “cd”

2. **Sum the frequencies**

Diagram:

- Node: r (frequency 3)
- Node: d (frequency 2)
- Node: c (frequency 1)
- Node: d (frequency 2)
- Merged node: rd (frequency 3)
Building the Huffman Tree

Step 2: Build a tree from the bottom up. Start by taking the two least frequent characters and merge them (create a parent node). Store the merged characters in a new queue.

Input:
\[ r : 1 \mid d : 2 \mid f : 2 \mid m : 2 \mid o : 3 \mid 'SPACE' : 3 \mid e : 4 \]

Output:

Single: \[ f : 2 \mid m : 2 \mid o : 3 \mid 'SPACE' : 3 \mid e : 4 \]

Merged: \( \text{rd} : 3 \)
Building the Huffman Tree

Step 3: Repeatedly merge the minimum two items from either list. Be sure to **remove and return** the minimum item as seen below:

**Input:**

**Single:** f: 2 | m: 2 | o: 3 | 'SPACE': 3 | e: 4

**Merged:** rd: 3

1st and smallest

3rd 4th smallest

Queue is easy to maintain!
Building the Huffman Tree

**Step 3:** Repeatedly merge the minimum two items from either list. Be sure to **remove and return** the minimum item as seen below:

**Input:**

**Single:** f: 2 | m: 2 | o: 3 | 'SPACE': 3 | e: 4

**Merged:** rd: 3

**Output:**

**Single:** o: 3 | 'SPACE': 3 | e: 4

**Merged:** rd: 3 | fm: 4
Building the Huffman Tree

**Step 3:** Repeatedly merge the minimum two items. Note that by **inserting in the back** the merged items will always remain sorted!

**Input:**

**Single:** o : 3 | 'SPACE' : 3 | e : 4

**Merged:** rd : 3 | fm : 4
Building the Huffman Tree

**Step 3:** Repeatedly merge the minimum two items. Note that by inserting in the back the merged items will always remain sorted!

**Input:**

**Single:** o : 3 | 'SPACE': 3 | e : 4

**Merged:** rd : 3 | fm : 4

**Output:**

**Single:** e : 4

**Merged:** rd : 3 | fm : 4 | o'SPACE': 6
Building the Huffman Tree

**Step 3:** Once the ‘single’ character list has been exhausted, we can easily merge the rest of our list by taking the front two values in merged.

**Input:**

**Single:** e : 4

**Merged:** rd : 3 | fm : 4 | o’SPACE’ : 6

**Output:**

**Single:**

**Merged:** fm : 4 | o’SPACE’ : 6 | rde : 7
Building the Huffman Tree

**Step 3:** Once the ‘single’ character list has been exhausted, we can easily merge the rest of our list by taking the front two values in merged.

**Input:**

**Single:**

**Merged:** fm : 4 | o’SPACE’ : 6 | rde : 7

**Output:**

**Single:**

**Merged:** rde : 7 | fmo’SPACE’ : 10
Building the Huffman Tree

**Step 4:** Stop when there is only a single item in either queue. This is our complete binary tree!

**Input:**

**Single:**

**Merged:** rde : 7 | fmo’SPACE’ : 10

**Output:**

**Single:**

**Merged:** rdefmo’SPACE’ : 17
Encoding using the Huffman Tree

The path through the tree defines each individual character’s encoding!

<table>
<thead>
<tr>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td></td>
</tr>
<tr>
<td>‘</td>
<td></td>
</tr>
<tr>
<td>‘</td>
<td></td>
</tr>
</tbody>
</table>
Encoding using the Huffman Tree

The path through the tree defines each individual character’s encoding!

<table>
<thead>
<tr>
<th>Char</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>100</td>
</tr>
<tr>
<td>e</td>
<td>01</td>
</tr>
<tr>
<td>d</td>
<td>001</td>
</tr>
<tr>
<td>m</td>
<td>101</td>
</tr>
<tr>
<td>r</td>
<td>000</td>
</tr>
<tr>
<td>o</td>
<td>110</td>
</tr>
</tbody>
</table>

' '    111
Decoding using the Huffman Tree

We can decode by walking through the tree using 0s and 1s as instructions!

**Input:** 100010100111110101

**Output:**