L20: Binary Heaps

Alex Steiger

CompSci 201: Spring 2024

3/27/2024
People in CS: Clarence “Skip” Ellis

- Born 1943 in Chicago. PhD in CS from UIUC in 1969
  - First African American in US to complete a PhD in CS
- Founding member of the CS department at U. Colorado, also worked in industry.
  - Developing original graphical user interfaces, object-oriented programming, collaboration tools.

“People put together an image of what I was supposed to be,” he recalled. “So I always tell my students to push.”

Read more here
Logistics, Coming up

• Today, Wednesday 3/29
  • APT 7 due

• Next Monday, 4/3
  • Nothing due, start on P5 Huffman

• Next Wednesday, 4/5
  • APT 8 due
Today’s agenda

• Wrap up Huffman Coding Intro

• Priority Queue revisited
  • Implementations, especially binary heap
Huffman Compression

Representing data with bits: Preferably fewer bits

- Zip
- Unicode
- JPEG
- MP3

Huffman compression used in all of these and more!
Decoding Variable Length

• What if we use
  • $a = 1$
  • $b = 10$
  • $c = 11$

• How would we decode 1011?
  • “baa” or “bc?”

• Problem: Encoding of $a$ (1) is a prefix of the encoding for $c$ (11). Ambiguous!

3/25/2024
CompSci 201, Spring 2024, Greedy & Huffman
Prefix Property: Encoding as a Tree

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>' '</td>
<td>001</td>
</tr>
</tbody>
</table>

Convention: 0 for left and 1 for right

Encoding is the sequence of 0’s and 1’s on root to leaf path

Values you want to encode are leaves: Ensures prefix property.

Values deeper in tree encoded with more bits than those earlier in the tree.
Decoding bits using Huffman tree

Goal: Decode 10011011 assuming it was encoded with this tree.

- Read bit at a time, traverse left or right edge.
- When you reach a leaf, decode the character, restart at root.

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>' '</td>
<td>001</td>
</tr>
</tbody>
</table>
Decoding bits using Huffman tree

Decode 10011011

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>' '</td>
<td>001</td>
</tr>
</tbody>
</table>

Initialize at root
Decoding bits using Huffman tree

Decode 10011011

Read 1, go to right child

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>' '</td>
<td>001</td>
</tr>
</tbody>
</table>
Decoding bits using Huffman tree

Decode 10011011

Read 0, go to left child

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>' '</td>
<td>001</td>
</tr>
</tbody>
</table>
Decoding bits using Huffman tree

Decode 10011011

Leaf, decode ‘g’, restart at root

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>' '</td>
<td>001</td>
</tr>
</tbody>
</table>
Decoding bits using Huffman tree

Decode 10011011

```
Read 0, go to left child
```

<table>
<thead>
<tr>
<th>char binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
</tr>
<tr>
<td>'o'</td>
</tr>
<tr>
<td>'p'</td>
</tr>
<tr>
<td>'h'</td>
</tr>
<tr>
<td>'e'</td>
</tr>
<tr>
<td>'r'</td>
</tr>
<tr>
<td>'s'</td>
</tr>
<tr>
<td>' '</td>
</tr>
</tbody>
</table>
Decoding bits using Huffman tree

Decode 10011011

```
g
```

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>' '</td>
<td>001</td>
</tr>
</tbody>
</table>

Read 1, go to right child
Decoding bits using Huffman tree

Decode 10011011

Read 1, go to right child
Decoding bits using Huffman tree

Decode 10011011

Read 0, go to left child

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>'.'</td>
<td>001</td>
</tr>
</tbody>
</table>
Decoding bits using Huffman tree

Decode 10011011

Leaf, decode ‘e’, restart at root

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>' '</td>
<td>001</td>
</tr>
</tbody>
</table>
Decoding bits using Huffman tree

Decode 10011011

ge

Read 1, go to right child

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>','</td>
<td>001</td>
</tr>
</tbody>
</table>

Compsci 201, Spring 2023, L20: Binary Heaps
Decoding bits using Huffman tree

Decode 10011011

ge

Read 1, go to right child

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>10</td>
</tr>
<tr>
<td>'o'</td>
<td>11</td>
</tr>
<tr>
<td>'p'</td>
<td>0100</td>
</tr>
<tr>
<td>'h'</td>
<td>0101</td>
</tr>
<tr>
<td>'e'</td>
<td>0110</td>
</tr>
<tr>
<td>'r'</td>
<td>0111</td>
</tr>
<tr>
<td>'s'</td>
<td>000</td>
</tr>
<tr>
<td>'.'</td>
<td>001</td>
</tr>
</tbody>
</table>
Decoding bits using Huffman tree

Decode 1001 1011

geo

Leaf, decode 'o'

<table>
<thead>
<tr>
<th>char binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
</tr>
<tr>
<td>'o'</td>
</tr>
<tr>
<td>'p'</td>
</tr>
<tr>
<td>'h'</td>
</tr>
<tr>
<td>'e'</td>
</tr>
<tr>
<td>'r'</td>
</tr>
<tr>
<td>'s'</td>
</tr>
<tr>
<td>' '</td>
</tr>
</tbody>
</table>
Huffman Coding

• **Greedy** algorithm for building an optimal variable-length encoding tree.

• High level idea:
  • Start with the leaves/values you want to encode with weights = frequency. Then repeat until all leaves are in single tree:
    • **Greedy step**: Choose the *lowest-weight nodes* to connect as children to a new node with weight = sum of children.

• Implementation? Use a priority queue!
Visualizing the greedy algorithm

Encoding the text “go go gophers”

<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>00</td>
</tr>
<tr>
<td>'o'</td>
<td>01</td>
</tr>
<tr>
<td>'p'</td>
<td>1110</td>
</tr>
<tr>
<td>'h'</td>
<td>1101</td>
</tr>
<tr>
<td>'e'</td>
<td>101</td>
</tr>
<tr>
<td>'r'</td>
<td>1111</td>
</tr>
<tr>
<td>'s'</td>
<td>1100</td>
</tr>
<tr>
<td>' '</td>
<td>100</td>
</tr>
</tbody>
</table>
Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

1

NetID * [ ]

solutions
Given the Huffman coding tree shown, what is the decoded text corresponding to the compressed bit sequence "1101 0111 1111 0010 1"?

These bits have been shown in blocks of 4 for readability; that does not mean each 4 bits codes for a single character.

```
* horse
```

Given these frequencies, how long will the encoding for 'a' be? How long will the encoding for 'b' be?

```
<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>30</td>
</tr>
<tr>
<td>b</td>
<td>20</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>15</td>
</tr>
<tr>
<td>e</td>
<td>40</td>
</tr>
</tbody>
</table>
```

- 'a' -> 1 bit, 'b' -> 1 bit
- 'a' -> 1 bits, 'b' -> 2 bits
- 'a' -> 2 bits, 'b' -> 2 bits
Suppose you are compressing a document with N total characters and M unique characters. How many nodes will there be in the Huffman coding tree? *

- O(N)
- O(N) (Correct)
- O(N + M)
- O(N log(N))
- O(M log(M))
- O(N^2)
- O(M^2)
P5 Outline

1. Write Decompress first
   • Takes a compressed file (we give you some)
   • Reads Huffman tree from bits
   • Uses tree to decode bits to text

2. Write Compress second
   • Count frequencies of values/characters
   • Greedy algorithm to build Huffman tree
   • Save tree and file encoded as bits
Priority Queues Revisited, Binary Heaps
java.util.PriorityQueue Class

- Kept in sorted order, smallest out first
- Objects must be Comparable OR provide Comparator to priority queue

```java
PriorityQueue<String> pq = new PriorityQueue<>();
pq.add("is");
pq.add("Compsci 201");
pq.add("wonderful");
while (! pq.isEmpty()) {
    System.out.println(pq.remove());
}
```

```java
PriorityQueue<String> pq = new PriorityQueue<>(Comparator.comparing(String::length));
pq.add("is");
pq.add("Compsci 201");
pq.add("wonderful");
while (! pq.isEmpty()) {
    System.out.println(pq.remove());
}
is
wonderful
Compsci 201
```
### java.util PriorityQueue basic methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
<th>Runtime Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(element)</td>
<td>Add an element to the priority queue</td>
<td>O(log(N)) comparisons</td>
</tr>
<tr>
<td>remove()</td>
<td>Remove and return the minimal element</td>
<td>O(log(N)) comparisons</td>
</tr>
<tr>
<td>peek()</td>
<td>Return (do <em>not</em> remove) the minimal element</td>
<td>O(1)</td>
</tr>
<tr>
<td>size()</td>
<td>Return number of elements</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Binary Heap at a high level

A **binary heap** is a binary tree satisfying the following structural invariants:

- **heap property**: every node is less than or equal to its successors, and
- **shape property**: the tree is **complete** (full except possibly last level, in which case it should be filled from left to right)
How are binary heaps typically implemented?

• Normally think about a conceptual binary tree underlying the binary heap.

• Usually implement with an array
  • minimizes storage (no explicit points/nodes)
  • simpler to code, no explicit tree traversal
  • faster too (constant factor, not asymptotically)---children are located by index/position in array
Aside: How much less memory?

- Storing an int takes 4 bytes = 32 bits on most machines.
- Storing one reference to an object (a memory location) takes 8 bytes = 64 bits on most machines.

- For a heap storing N integers...
  - Array of N integers takes ~ 4N bytes.
  - Binary tree where each node has an int, left, and right reference takes ~20N bytes.
  - So maybe a 5x savings in memory (just an estimate). \textit{Not} an asymptotic improvement.
Using an array for a Heap

- Makes it easy to keep track of last “node” in “tree”
- Index positions in the tree level by level, left to right:

```
Depth 0
Depth 1
Depth 2
Depth 3
```

- Last node in the heap is always just the largest non-empty index
- Can use indices to represent as an array!

```
<table>
<thead>
<tr>
<th>6</th>
<th>10</th>
<th>7</th>
<th>17</th>
<th>13</th>
<th>9</th>
<th>21</th>
<th>19</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
```

(ArrayList if you want it to be growable)
Properties of the Heap Array

- Store “node values” in array beginning at index 1
  - Could 0-index, Zybook does this
- Last “node” is always at the max index
- Minimum “node” is always at index 1
- **peek** is easy, return first value.
  - How about add?
  - Remove?
Relating Nodes in Heap Array

- When 1-indexing: For node with index $k$
  - left child: index $2k$
  - right child: index $2k + 1$
  - parent: index $k/2$

- Why? Follows from:
  - Heap is complete, and
  - Complete binary tree has $2^d$ nodes at depth $d$ (except last level)
Adding values to heap in pictures

• Add to first open position in last level of the tree
  • (really, add to end of array)
• Shape property satisfied, but not heap property
• Fix it: Swap with parent if heap property violated
  • Stop when parent is smaller,
  • or you reach the root
Heap add implementation

```java
public void add(Integer value) {
    heap.add(value); // add to last position
    size++;

    int index = size; // note we are 1-indexing
    int parent = index / 2;

    while (parent >= 1 && heap.get(parent) > heap.get(index)) {
        swap(index, parent);
        index = parent;
        parent /= 2;
    }
}
```

```
ArrayList<Integer> heap

index=10
parent=5
```

```
6 10 7 17 13 9 21 19 25 8
0 1 2 3 4 5 6 7 8 9 10
```
Heap add implementation

```java
public void add(Integer value) {
    heap.add(value); // add to last position
    size++;
    int index = size; // note we are 1-indexing
    int parent = index / 2;
    while (parent >= 1 && heap.get(parent) > heap.get(index)) {
        swap(index, parent);
        index = parent;
        parent /= 2;
    }
}
```

**ArrayList<Integer> heap**
Heap add implementation

```java
public void add(Integer value) {
    heap.add(value); // add to last position
    size++;
    int index = size; // note we are 1-indexing
    int parent = index / 2;
    while (parent >= 1 && heap.get(parent) > heap.get(index)) {
        swap(index, parent);
        index = parent;
        parent /= 2;
    }
}
```
Heap remove in pictures

- Always return root value
- How to repair shape into a single tree?
  - Replace root with last node in the heap
  - While heap property violated, swap with *smaller* child.
Heap remove implementation

```java
public Integer remove() {
    if (size < 1) { return null; }
    Integer retVal = heap.get(index:1);
    heap.set(index:1, heap.get(size));
    heap.remove(size);
    size--;
    if (size == 0) { return retVal; }
```

Get the minimal value
Replace "root" with "last node"
Delete "last node"

![Heap diagram before and after removal](image)
Heap remove implementation

```java
46  int index = 1;
47  int minChild = 2;
48  if (size > 2 && heap.get(index:3) < heap.get(index:2)) { minChild = 3; }
49  while (minChild <= size && heap.get(index) > heap.get(minChild)) {
50      swap(index, minChild);
51      index = minChild;
52      minChild = minChild * 2;
53      if (size > minChild && heap.get(minChild + 1) < heap.get(minChild)) { minChild++; }
54  }
55  return retVal;
```

Find the smaller of 2 child nodes

Swap

Violating heap property

minChild

Find the smaller of 2 child nodes

Swap

Violating heap property

minChild
Heap remove implementation

```c
46    int index = 1;
47    int minChild = 2;
48    if (size > 2 && heap.get(index:3) < heap.get(index:2)) { minChild = 3; }
49    while (minChild <= size && heap.get(index) > heap.get(minChild)) {
50        swap(index, minChild);
51        index = minChild;
52        minChild = minChild * 2;
53        if (size > minChild && heap.get(minChild + 1) < heap.get(minChild)) { minChild++; }
54    }
55    return retVal;
```
Heap remove implementation

```python
int index = 1;
int minChild = 2;
if (size > 2 && heap.get(index:3) < heap.get(index:2)) { minChild = 3; }
while (minChild <= size && heap.get(index) > heap.get(minChild)) {
    swap(index, minChild);
    index = minChild;
    minChild = minChild * 2;
    if (size > minChild && heap.get(minChild + 1) < heap.get(minChild)) { minChild++; }
}
return retVal;
```

Return retVal (6)
Heap Complexity

- Claimed that:
  - Peek: $O(1)$
  - Add: $O(\log(N))$
  - Remove: $O(\log(N))$

- On a heap with $N$ values. Why?
  - Peek: Easy, return first value in an Array
  - Complete binary tree always has height $O(\log(N))$.
  - .add and remove “traverse” one root-leaf path, length at most $O(\log(N))$. 
decreaseKey Operation?

• Suppose we decrease the 13 to 5.
• Violates heap property
• Fix like in the add operation:
  • While violating heap property, swap with parent
decreaseKey NOT in java.util

• decreaseKey is important for some algorithms, but not supported in many standard libraries (including the java.util.PriorityQueue)

• Why not?
  • Note that binary heap does not support efficient search
  • In order to do decreaseKey in $O(\log(n))$ time, need to store references/indices of all the “nodes.”
  • Adds overhead, not done in java.util
Alternative Implementation: Binary Search Tree

• If your keys happen to be unique…
• Can support $O(\log(n))$ add & remove (smallest) using a binary search tree!
• Smallest is leftmost child
PriorityQueue (with unique keys) using a java.util TreeSet

import java.util.TreeSet;

public class BSTPQ<T extends Comparable<T>> {
    private TreeSet<T> bst;

    public BSTPQ() { bst = new TreeSet<>(); }
    public void add(T element) { bst.add(element); }
    public int size() { return bst.size(); }
    public T peek() { return bst.first(); }

    public T remove() {
        T returnValue = bst.first();
        bst.remove(returnValue);
        return returnValue;
    }

    public void decreaseKey(T oldKey, T newKey) {
        bst.remove(oldKey);
        bst.add(newKey);
    }
}

first gives smallest element in TreeSet in O(log(n)) time

Can decreaseKey by removing and then re-adding, both O(log(n)) time for a TreeSet
Disadvantages to using a Binary Search Tree for your priority queue?

1. All elements must be unique

2. Not array-based, uses more memory and has higher constant factors on runtime

3. Much harder to implement with guarantees that the tree will be balanced.