CSE100

Advanced Data Structures

Lecture 13

(Based on Paul Kube course materials)
CSE 100

- Priority Queues in Huffman’s algorithm
- Heaps and Priority Queues
- Time and space costs of coding with Huffman codes

Reading: Weiss Ch. 6, Ch. 10.1.2
Priority queues and Huffman’s algorithm

• The data structure holding the Huffman forest needs to support these operations:
  • Create an initially empty structure
  • Insert a tree into the structure
  • Delete from the structure, and return, the tree with the smallest count in its root
• These are exactly the typical operations provided by the Priority Queue abstract data type, used in many fundamental algorithms:
  • Create an empty Priority Queue
  • Insert an item into the Priority Queue
  • Delete, and return, the item in the Priority Queue with highest priority
• A Priority Queue can be implemented in many ways... one of the best ways is to use a heap
Implementing a priority queue using a heap

• A heap is a binary tree with these properties:
  • **structural property**: each level is completely filled, with the possible exception of the last level, which is filled left-to-right
    (heaps are “complete” binary trees, and so are balanced: Height H = O(logN))
  • **ordering property**: for each node X in the heap, the key value stored in X is greater than or equal to all key values in descendants of X
    (this is sometimes called a MAX-heap; for MIN-heaps, replace greater than with less than)

• Heaps are often just called “priority queues”, because they are natural structures for implementing the Priority Queue ADT. They are also used to implement the heapsort sorting algorithm, which is a nice fast $N \log N$ sorting algorithm

• Note: This heap data structure has NOTHING to do with the “heap” region of machine memory where dynamic data is allocated...
The heap structural property

- These trees satisfy the structural property of heaps:

- These do not:
The heap ordering property

• This tree satisfy the ordering property of (max) heaps:

• This does not:
Inserting a key in a heap

- When inserting a key in a heap, must be careful to preserve the structural and ordering properties. The result must be a heap!
- Basic algorithm:
  1. Create a new node in the proper location, filling level left-to-right
  2. Put key to be inserted in this new node
  3. “Bubble up” the key toward the root, exchanging with key in parent, until it is in a node whose parent has a larger key (or it is in the root)

Insert in a heap with N nodes has time cost $O(\log N)$
Deleting the maximum key in a heap

- When deleting the maximum valued key from a heap, must be careful to preserve the structural and ordering properties. The result must be a heap!
- Basic algorithm:
  1. The key in the root is the maximum key. Save its value to return
  2. Identify the node to be deleted, unfilling bottom level right-to-left
  3. Move the key in the node to be deleted to the root, and delete the node
  4. “Trickle down” the key in the root toward the leaves, exchanging with key in the larger child, until it is in a node whose children have smaller keys, or it is in a leaf

- deleteMAX in a MAX heap is $O(\log N)$.
  (What is deleteMIN in a MAX heap?)
Analysis of Huffman’s algorithm with heap priority queue

- Suppose the input source is a sequence of length K drawn from an alphabet of N symbols

Step 1: time cost $O(K)$
Step 2: $O(N \log N)$ (can actually be done in $O(N)$ if you are a bit clever)
Step 3: the loop is executed N-1 times
Step 3a: time cost $O(\log N)$
Step 3b: time cost $O(1)$
Step 3c: time cost $O(\log N)$
Step 4: time cost $O(1)$

- So overall time cost: $O(K + N \log N)$
- (Compare to $O(K + N^2)$ when using a linked list priority queue)
- But if K is large, or slow I/O is involved, Step 1 will dominate and choice of priority queue implementation may not matter significantly
Time cost of Huffman coding

- We have analyzed the time cost of constructing the Huffman code tree.
- What is the time cost of then using that tree to code the input message?
- Coding one symbol from the input message takes time proportional to the number of bits in the code for that symbol; and so the total time cost for coding the entire message is proportional to the total number of bits in the coded version of the message.
- If there are \( K \) symbols in the input message, and the average number of bits per symbol in the Huffman code is \( H' \), then the time cost for coding the message is proportional to \( H' \cdot K \).
- What can we say about \( H' \)? This requires a little information theory...
Sources of information and entropy

• A source of information emits a sequence of symbols drawn independently from some alphabet
• Suppose the alphabet is the set of symbols \( \{ \sigma_1, \ldots, \sigma_N \} \)
• Suppose the probability of symbol \( \sigma_i \) occurring in the source is \( p_i \)
• Then the information contained in symbol \( \sigma_i \) is \( \log \frac{1}{p_i} \) bits, and the average information per symbol is (logs are base 2):

\[
H = \sum_{i=1}^{N} p_i \log \frac{1}{p_i}
\]
• This quantity \( H \) is the “entropy” or “Shannon information” of the information source
• For example, suppose a source uses 3 symbols, which occur with probabilities 1/3, 1/4, 5/12
• The entropy of this source is

\[
\frac{1}{3} \log 3 + \frac{1}{4} \log 4 + \frac{5}{12} \log \frac{12}{5} \approx 1.5546 \text{bits}
\]
How large and how small can entropy be?

- A source of information emits a sequence of symbols drawn independently from the alphabet \(\{\sigma_1, \ldots, \sigma_N\}\) such that the probability of symbol \(\sigma_i\) occurring is \(p_i\).
- The entropy (Shannon information) of the source, in bits, is defined as (logs are base 2):

\[
H = \sum_{i=1}^{N} p_i \log \frac{1}{p_i}
\]

- Q: What is the possible range of values of \(H\)? A: We always have \(0 \leq H \leq \log N\).
- The smallest possible value of \(H\) is 0:
  - If one symbol \(\sigma_i\) occurs all the time, so \(p_i = 1\) and so \(\log \frac{1}{p_i}\), and all the other symbols \(\sigma_j\) never occur, so the other \(p_i = 0\), then you don’t get any information by observing the source:

\[
H = 0
\]
- The largest possible value of \(H\) is \(\log N\). This is the ‘maximum entropy’ condition
  - If each of the symbols are equally likely, then \(p_i = \frac{1}{N}\) for all \(i\) and so:

\[
H = \sum_{i=1}^{N} \frac{1}{N} \log N = \log N
\]
Entropy and coding sources of information

- A uniquely decodable code for an information source is a function that assigns to each source symbol $\sigma_i$ a unique sequence of bits, called the code for $\sigma_i$.
- Suppose the number of bits in the code for symbol $\sigma_i$ is $n_i$.
- Then the average number of bits in the code for the information source is

$$H' = \sum_{i=1}^{N} p_i n_i$$

- Let $H'_{\text{min}}$ be the average number of bits in the code with the smallest average number of bits of all uniquely decodable codes for an information source with entropy $H$. Then you can prove the following interesting results:
  - $H \leq H'_{\text{min}} < H + 1$, i.e. you can do no better than $H$, and will never do worse by more than one bit per symbol.
  - The Huffman code for this source has codes with average number of bits $H'_{\text{min}}$, i.e., there is no better uniquely decodable code than Huffman.
  - So Huffman-coding a message consisting of a sequence of $K$ symbols obeying the probabilities of an information source with entropy $H$ takes at least $H' \cdot K$ bits but less than $H' \cdot K + K$ bits.
Entropy and Huffman: an example

- Suppose the alphabet consists of the 8 symbols A, B, C, D, E, F, G, H, and the message sequence to be coded is AAAAAABBAHBCBGC

- Given an alphabet of 8 symbols, the entropy $H$ of a source using those symbols must lie in the range $0 \leq H \leq \log_2 N = 3$. But if the source emits symbols with probabilities as evidenced in this message, what exactly is its entropy $H$?

- The table of counts or frequencies of symbols in this message would be:
  - A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2

- The message has length 17, and so the probabilities of these symbols are:
  - A: 6/17; B: 4/17; C: 4/17; D: 0; E: 0; F: 0; G: 1/17; H: 2/17

- The entropy of this message (and any message with those symbol probabilities) is then

$$
\frac{6}{17} \log \frac{17}{6} + \frac{4}{17} \log \frac{17}{4} + \frac{4}{17} \log \frac{17}{4} + 0 + 0 + 0 + \frac{1}{17} \log 17 + \frac{2}{17} \log \frac{17}{2} \approx 2.1163 \text{ bits}
$$

- As we saw last time, the Huffman code for this message gives $H'_{\text{min}} = 37/17 \approx 2.1768$ bits per symbol, which is quite close to the message entropy.
Huffman code trees, using dynamic data and pointers

- A class definition for a Huffman code tree node might have member variables along these lines (assume the symbols to be coded come from an alphabet of no more than 256 items):

```cpp
class HCNode {
    HCNode* parent;  // pointer to parent; null if root
    bool isChild0;  // true if this is "0" child of its parent
    HCNode* child0;  // pointer to "0" child; null if leaf
    HCNode* child1;  // pointer to "1" child; null if leaf
    unsigned char symb;  // symbol
    int count;  // count/frequency of symbols in subtree
};
```

- The class definition could also have some methods and constructors for common operations and initialization of `HCNode` objects
- When building a Huffman code tree, the fields of the parent and child nodes need to be set appropriately
- The result is a tree data structure that is useful both for coding and decoding
- (For coding, you should also have an array or other table structure of pointers to code tree leaf nodes, indexed by data items to code)
Implementing trees using arrays

- Arrays can be used to implement trees; this is a good choice in some cases if the tree has a very regular structure, or a fixed size.
- One common case is: implementing a heap.
- Because heaps have a very regular structure (they are complete binary trees), the array representation can be very compact: array entries themselves don’t need to hold parent-child relational information.
  - The index of the parent, left child, or right child of any node in the array can be found by simple computations on that node’s index.
- Huffman code trees do not have as regular a structure as a heap (they can be extremely unbalanced), but they do have some structure, and they do have a determinable size.
  - Structure: they are “full” binary trees; every node is either a leaf, or has 2 children.
  - Size: to code N possible items, they have N leaves, and N-1 internal nodes.
- These features make it potentially interesting to use arrays to implement a Huffman code tree.
Implementing heaps using arrays

- The heap structural property permits a neat array-based implementation of a heap
- Recall that in a heap each level is filled left-to-right, and each level is completely filled before the next level is begun (heaps are "complete" binary trees)
- One way to implement a heap with $N$ nodes holding keys of type $T$, is to use an $N$ element array $T$ heap[$N$]; Nodes of the heap will correspond to entries in the array as follows:
  - The root of the heap is array element indexed 0, heap[0]
  - if a heap node corresponds to array element indexed $i$, then
    - its left child corresponds to element indexed $2*i + 1$
    - its right child corresponds to element indexed $2*i + 2$
    - ..and so a node indexed $k$ has parent indexed $(k-1)/2$
- The result: Nodes at the same level are contiguous in the array, and the array has no "gaps"
- Now it is easy to implement the heap Insert and Delete operations (in terms of "bubbling up" and "trickling down"); and easy to implement $O(N \log N)$ “heap sort”, in place, in a $N$-element array
Implementing heaps using arrays: a picture

- To map out the correspondence between a tree-diagram and array version of a heap, follow the formulas, or follow the rule for the structural property of a heap: levels are filled left to right, and top to bottom.
Priority queues in C++

- It is not too hard to code up your own priority queue, and using the heap-structured array is a nice way to do it.
- But the C++ STL has a `priority_queue` class template, which provides functions `push()`, `top()`, `pop()`, `size()`.
- A C++ `priority_queue` is a generic container, and can hold any kind of thing as specified with a template parameter when it is created: for example `HCNode`s, or pointers to `HCNode`s, etc.
  ```c++
  #include <queue>
  std::priority_queue<HCNode> p;
  ```

- However, objects in a priority queue must be comparable to each other for priority.
- By default, a `priority_queue<T>` uses `operator<` defined for objects of type `T`.
  - if `a < b`, `b` is taken to have higher priority than `a`.
- So let’s see how to override that operator for `HCNode`s.
Overriding operator< for HCNodes: header file

In a header file, say HCNode.hpp:

```cpp
#ifndef HCNODE_HPP
#define HCNODE_HPP

class HCNode {
    friend class HCTree; // so an HCTree can access HCNode fields

private:
    HCNode* parent;       // pointer to parent; null if root
    bool isChild0;        // true if this is "0" child of its parent
    HCNode* child0;       // pointer to "0" child; null if leaf
    HCNode* child1;       // pointer to "1" child; null if leaf
    unsigned char symb;   // symbol
    int count;            // count/frequency of symbols in subtree

public:
    // for less-than comparisons between HCNodes
    bool operator<(HCNode const &) const;
};
#endif
```
Overriding operator< for HCNodes: implementation file

In an implementation file, say HCNode.cpp:

```cpp
#include "HCNode.hpp"
/** Compare this HCNode and other for priority ordering.
 * Smaller count means higher priority.
 * Use node symbol for deterministic tiebreaking
 */

bool HCNode::operator<(HCNode const & other) const {
    // if counts are different, just compare counts
    if(count != other.count) return count > other.count;
    // counts are equal. use symbol value to break tie.
    // (for this to work, internal HCNodes must have symb set!)
    return symb < other.symb;
}
```
Using operator< to compare HCNodes

• Consider this context:

```c
HCNode n1, n2, n3, n4;
n1.count = 100; n1.symb = ’A’;
n2.count = 200; n2.symb = ’B’;
n3.count = 100; n3.symb = ’C’;
n4.count = 100; n4.symb = ’A’;
```

• Now what is the value of these expressions?

```c
n1 < n2
n2 < n1
n2 < n3
n1 < n3
n3 < n1
n1 < n4
```
Using std::priority_queue in Huffman’s algorithm

- If you create an STL container such as priority_queue to hold HCNode objects:
  
  ```
  #include <queue>
  std::priority_queue<HCNode> pq;
  ```

- ... then adding an HCNode object to the priority_queue:
  
  ```
  HCNode n;
  pq.push(n);
  ```

- ... actually creates a copy of the HCNode, and adds the copy to the queue. You probably don’t want that. Instead, set up the container to hold pointers to HCNode objects:
  
  ```
  std::priority_queue<HCNode*> pq;
  HCNode* p = new HCNode();
  pq.push(p);
  ```

- But there’s a problem: our operator< is a member function of the HCNode class. It is not defined for pointers to HCNodes. What to do?
std::priority_queue template arguments

- The template for priority_queue takes 3 arguments:
  
  ```cpp
template < class T,  
     class Container = vector<T>,  
     class Compare = less<typename Container::value_type> >  
  class priority_queue
  ```

- The first is the type of the elements contained in the queue.
- If it is the only template argument used, the remaining 2 get their default values:
  - a `vector<T>` is used as the internal store for the queue,
  - `operator<` is used for priority comparisons
- Using a vector for the internal store is fine, but we want to tell the priority_queue to first dereference the HCNode pointers it contains, and then apply `operator<`
- How to do that?
Defining a "comparison class"

• The documentation says of the third template argument:
  • Compare: Comparison class: A class such that the expression comp(a,b), where comp is an object of this class and a and b are elements of the container, returns true if a is to be placed earlier than b in a strict weak ordering operation. This can be a class implementing a function call operator...

• Here’s how to define a class implementing the function call operator operator() that performs the required comparison:

```cpp
class HCNodePtrComp {
    bool operator()(HCNode* & lhs, HCNode* & rhs) const {
        // dereference the pointers and use operator<
        return *lhs < *rhs;
    }
};
```

• Now, create the priority_queue as:

```cpp
std::priority_queue<HCNode*,std::vector<HCNode*>,HCNodePtrComp> pq;
```
• and priority comparisons will be done as appropriate.
Huffman code trees using arrays: one approach

- Suppose, for example, that the items to be coded are bytes: 8-bit integers
- A Huffman code tree for this domain will have (no more than) 256 leaves, and (no more than) 255 internal nodes
- We could represent the tree using arrays along these lines:
  ```
  // this array will hold indices of parents of leaf nodes
  unsigned char leaves[256];
  // this array will hold indices of parents of internal nodes
  unsigned char intnodes[255];
  ```

- Now given the value of an unsigned char `c` to code, `leaves[c]` gives the index of its parent in the `intnodes` array
- `... intnodes[leaves[c]]` gives the index of the parent of that node in the `intnodes` array, etc.
  - (Note the `unsigned char` type in C++ has values in the range 0 through 255 inclusive)
- The root of the tree can be indicated by an array entry with value 255 (its parent is not in the `intnodes` array...)
“0” child or “1” child?

• That array representation will indicate which nodes are the children of which other nodes in the tree...
  • But it does not indicate whether a child is a “0” child or a “1” child, and this is essential for the coding tree
• We could follow a convention such as: of the two children, the one with smaller index is always the “1” child (and all leaf node indices are considered smaller than all internal node indices)
• ... Or, we can represent these additional “bits” of information about the nodes using two bit sequences: one with 256 bits for the leaf nodes, and one with 255 bits for the internal nodes
• These bit sequences can be packed into 32 bytes each:

// this array holds bits indicating what kind of child a leaf is
char leafchildbit[32];
// this array holds bits indicating what kind of child a nonleaf is
char intnodechildbit[32];

• For example: if bit 65 is “1” in the leafchildbit array, that indicates that the leaf node with data value 65 is a “1” child of its parent
Huffman code trees using arrays, cont’d

• Let’s see how these arrays would look when representing the Huffman code tree we have been using as an example

• Note that the ASCII code values of characters A,B,C,G,H are 65,66,67,71,72 respectively

• See picture on next page...
A picture: an array representation of a Huffman tree

leaves

<table>
<thead>
<tr>
<th>...</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
</tr>
</tbody>
</table>

intnodes

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>3</th>
<th>255</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
</tr>
</tbody>
</table>

leafchildbit: bits 65, 67, 71 are 1, others 0
intnodechildbit: bit 2 is 1, others 0
About this array representation

- This array representation scheme faithfully and completely represents the Huffman code tree, and is useful as a coding tree (easy to follow links from leaf to root)

- But the scheme does not explicitly indicate for a given node what the children of that node are... (that information is implicit, not explicit, in that scheme)

- To be useful as a tree for efficient decoding, the representation could be augmented with arrays that explicitly indicate, for each node, what nodes are its “1” and “0” children, together with an explicit index of the root node; or it could be converted into such a representation

- However, as it is, this scheme is a compact representation of the entire tree; it permits representing any Huffman code on a 256-symbol alphabet using only 256 + 255 + 32 + 32 = 575 bytes

- Recall that enough information to reconstruct the Huffman coding tree should be stored in a Huffman-compressed file; these arrays (or a variant of them) would be one way of doing that
Huffman code trees using arrays: another approach

- The approach just sketched made no use of the special structure of Huffman code trees, other than the fact that if there are $N$ leaf nodes, there are $N-1$ internal nodes.
- Using the fact that the Huffman tree is “full” -- every node is either a leaf or has 2 children -- permits another approach leading to a more compact representation.
- Consider a usual (inorder or preorder or postorder) traversal of a binary tree: it visits the leaves in left-to-right order.
- Suppose when visiting each leaf, its level in the tree is recorded.
- This sequence of left-to-right leaf levels does not specify the structure of the tree for arbitrary binary trees... but it does for full binary trees!
- For example, the left-to-right leaf level sequence 1,2,4,4,3 can be obtained only from this full binary tree:
Huffman code trees using arrays: another approach, cont’d

- So, you can uniquely reconstruct a Huffman code trie with the following information:
  - the left-to-right sequence of leaf levels
  - the corresponding sequence of message symbols in those leaves
  - ... together with an assumption about which (left or right) children are “0” children

- For example: the level sequence 1,2,4,4,3 and the symbol sequence A,B,G,H,C and the assumption that left children are “0” children uniquely determine this Huffman trie:

- Assuming a 256-symbol alphabet, leaf levels are in the range 0-255, and so with this approach any code trie could be represented using at most 256 + 256 bytes; or by using a byte to specify the value $m$ equal to how many of the 256 possible symbols actually appear in the message, the trie could be represented using $1 + 2m$ bytes

- Note that this representation isn’t directly useful for coding or decoding... you would use it to reconstruct the tree, and then use the tree
Representing the information in a Huffman code tree

- Huffman’s algorithm does construct a binary trie, which represents a coding scheme to use for coding and decoding messages.
- But for actually coding and decoding a message, it really isn’t required to use a binary trie.
- What you need is:
  - A way to go from message symbols to bit sequences, for coding a message.
  - A way to go from bit sequences to message symbols, for decoding a message.
- As we have seen, the Huffman code tree itself permits doing those things, but other data structures also can.
- For example:
  - An array of Strings of “1”s and “0”s, indexed by message symbol.
  - A HashMap in which keys are Strings of “1”s and “0”s, and values are message symbols.
- Other approaches are possible as well. “In computer science, if there is one way to do something, there are infinitely many ways to do it!”
Next time...

• C++ I/O
• Some useful classes in `<iostream>`
• I/O buffering
• Bit-by-bit I/O

Reading: online documentation on C++ streams