CSE100

Advanced Data Structures

Lecture 12

(Based on Paul Kube course materials)
CSE 100

- Coding and decoding with a Huffman coding tree
- Huffman coding tree implementation issues
- Priority queues and priority queue implementations

Reading: Weiss Ch. 6, Ch. 10.1.2
Huffman code trees

- Last time, we discussed the Huffman coding algorithm.
- The Huffman algorithm takes as input the probability of occurrence of each symbol in the alphabet used by an information source, and constructs a tree (a binary trie) which represents a code for that information source.
- This tree (or equivalent information) is used to code symbols from the information source; the code for each item is a sequence of bits determined by the tree.
- An identical tree (or equivalent information) must be used to decode that sequence of bits, to get back the original sequence of symbols that was coded!
- How can the tree be used to do those things?
A Huffman tree

• Suppose Huffman’s algorithm has constructed this coding tree:

• ... and you want to code the sequence of symbols:

  A A A A A B B A H H B C B G C C C
Using the code tree to code a message

- Once you have the Huffman code tree, coding the message sequence is easy:
  - For each symbol in the message, output the sequence of bits given by the 1’s and 0’s on the path from the root of the code tree to that symbol
    - (This can be done by starting at the leaf node containing the symbol, follow parent pointers to the root with recursive calls, then output the sequence of bits as the recursive calls pop from the runtime stack; or once the tree has been constructed, you can traverse it and build a lookup table of codes indexed by symbol)

- So, the message sequence A A A A A B B A H H B C B G C C C is coded as...

  ..... which is ______ bits

- (Recall that the naive 3-bit-per-symbol coding would take 51 bits)
Using the code tree to code a message

AAAAABBAHHBCBGCCC --&gt; 11
AAAAABBAHHBCBGCCC --&gt; 1111
AAAAABBAHHBCBGCCC --&gt; 111111
AAAAABBAHHBCBGCCC --&gt; 11111111
AAAAAABBAHHBCBGCCC --&gt; 11111111
AAAAAABBAHHBCBGCCC --&gt; 1111111111
AAAAAABBAHHBCBGCCC --&gt; 111111111110
AAAAAABBAHHBCBGCCC --&gt; 11111111111010
AAAAAABBAHHBCBGCCC --&gt; 1111111111101011
AAAAAABBAHHBCBGCCC --&gt; 1111111111101011000
AAAAAABBAHHBCBGCCC --&gt; 1111111111101011000000
AAAAAABBAHHBCBGCCC --&gt; 111111111110101100000010
AAAAAABBAHHBCBGCCC --&gt; 1111111111101011000000101
AAAAAABBAHHBCBGCCC --&gt; 11111111111010110000001011
AAAAAABBAHHBCBGCCC --&gt; 111111111110101100000010110
AAAAAABBAHHBCBGCCC --&gt; 1111111111101011000000101101
AAAAAABBAHHBCBGCCC --&gt; 11111111111010110000001011010
AAAAAABBAHHBCBGCCC --&gt; 111111111110101100000010110101
AAAAAABBAHHBCBGCCC --&gt; 1111111111101011000000101101010
AAAAAABBAHHBCBGCCC --&gt; 11111111111010110000001011010101
... 37 bits!
Using the code tree to decode a message

- With the Huffman code tree, decoding a coded message is also easy
  - Start with the first bit in the coded message, and start with the root of the code tree
  - As you read each bit, move to the left or right child of the current node, matching the bit just read with the label on the edge
  - When you reach a leaf, output the symbol stored in the leaf
  - Start at the root of the code tree again, and read the next bit

- Note: Many distinct Huffman codes exist for a given information source (for example, take any one Huffman tree, exchange “0” and “1” labels on children of any nodes, and get a different code). These are all optimal codes. But it is important to use exactly the same code for decoding as for encoding, or you won’t be able to reconstruct the input sequence

- The tree, or enough information to reconstruct the tree exactly, must be included before the beginning of the coded message (or in some other way transmitted to the receiver)
Remarks on Huffman codes

• The code constructed by Huffman’s algorithm is optimal (it uses the fewest possible bits to code the message) if the occurrences of symbols in the message are probabilistically independent, and the same code is used for a symbol no matter where it occurs
  • this independence doesn’t hold in many cases (ordinary English text, for example!) and in those cases other coding techniques can be better

• Huffman’s algorithm requires knowing the frequency of occurrence of symbols in the message to be coded, before the code can be constructed
  • other approaches (e.g. adaptive Huffman coding, or the LZW coding used in the zip and gzip utilities) adapt to the statistics in a message “on the fly”

• Like all coding or compression schemes that depend on the message to be coded, enough information must be included in the coded message to permit the message to be decoded (decompressed) later
  • this additional information can be the frequencies of symbols in the original (uncompressed) message, or a representation of the code tree itself
  • for some message sequences (short ones, ones already compressed, etc.) this additional required information may make the “compressed” version longer than the original!
Basic properties of a Huffman code tree

• Suppose there are \( N \) different symbols in the input sequence (some of them may appear more than once)
  • For example, the input sequence
    
    THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
    
    has length 43, and contains 27 different symbols: The capital letters A-Z, and space

• The Huffman trie-construction algorithm starts with a forest of \( N \) one-node trees, and then, while there is more than one tree in the forest:
  • removes two trees from the forest,
  • makes them the left and right subtrees of a new parent node, and
  • returns the new tree to the forest
• The result is a “full” binary tree: every non-leaf node has exactly 2 children (why?!) 

• Questions: in the resulting tree as functions of \( N \):
  • How many leaf nodes are there? _________
  • How many internal (non-leaf) nodes are there? _________
  • So, how many nodes in the code tree? _________
Coding with a Huffman tree

• Suppose you have this Huffman coding tree:

![Huffman tree diagram]

• ... and you want to code the letter A
• The code for letter A is the sequence of bits labeling edges on the path from the root of the tree to the leaf containing A
• How to find that path?
Coding with a Huffman tree, cont’d

- A not very good way to do it is to start at the root, and try to find a path to the leaf containing A

- (You *could* do that, but worst case you would have to traverse the entire tree before finally finding the leaf you want, taking $\Omega(N)$ steps)

- A better way: start at the leaf containing A, follow the (unique, since in a tree every non-root node has a unique parent) path to the root of the tree keeping track of the bits on the path... and then reverse that sequence of bits

- A clever way to do that uses recursion

- To see how to do that, we will take a short detour into linked lists...
Traversing a data structure

- A data structure contains elements, which contain data
- *Traversing* or *iterating over* a data structure means: “visiting” or “touching” the elements of the structure, and doing something with the data
- For example you could have a singly-linked list, with elements that are instances of this class:
  ```
  class LNode {
      int data;
      LNode* next;
  }
  ```
- Suppose `first` is a pointer that points to the first element of a list of LNode objects; the last element in the list has a null `next` field

```
first:  data: 1  next:  
        data: 2  next:  
        data: 3  next: /
```

- Traversing that list, from first to last, and printing out data in the elements, would print:
  1 2 3
Traversing a linked list

• Suppose `first` is a pointer that points to the first element of a long list of `LNode` objects; the last element in the list has a null `next` field

• To traverse it from first element to last, you could do it iteratively:

```cpp
void traverse(LNode* n) {
    while(n) {
        std::cout << n->data << std::endl;
        n = n->next;
    }
}
```

• Now `traverse(first)` will traverse the list pointed to by `first`, printing out the data fields, first element to last
  • (It is also possible to do this recursively)
Traversing a linked list, in reverse

- Again, suppose `first` is a pointer that points to the first element of a long list of `LNode` objects; the last element in the list has a null `next` field.
- But suppose now that you want to traverse it from last element to first.
- Here, it is possible to do it iteratively, but it is much neater to do it recursively:

```cpp
// PRE: n points to a node of a list with null next field
// in its last node; or n is null
// POST: data fields in the list from node n through the last
// node have been printed, in reverse order
void traverse(LNode* n) {
    if(!n) return;  // base case: empty list
    traverse(n->next);  // traverse the rest of the list, reversed
    std::cout << n->data << std::endl;  // print out this elt
}
```

- Now `traverse(first)` will traverse the entire list, last element to first.
- (What happens if the 2nd and 3rd statements in the method are exchanged?)
Root-to-leaf path traversals in a Huffman code tree

- Consider the nodes from a leaf to the root as a linked list
  - (this requires that each node have a link to its parent: a “parent pointer”)
  - thus, the whole tree can be thought of as a collection of linked lists, which share nodes and links

- Consider the leaf as the first element in the list
  - to find quickly the leaf corresponding to an item to be coded, maintain a table data structure that associates each item to be coded with a pointer to its leaf node in the tree

- Traverse this list, in reverse order. When a node is visited, output “0” or “1” depending on whether it is a “0” or “1” child of its parent
  - this requires that a node somehow specify whether it is a “0” child or a “1” child of its parent
Decoding with a Huffman tree

- Suppose you have this Huffman coding tree:

- ... and you want to decode the bit sequence 001
- The letter for that code is contained in the leaf you find at the end of the path starting at the root, following links to “0” or “1” children as given by the bit sequence
- How to find that path?
Decoding with a Huffman tree, cont’d

• Decoding with a Huffman tree is a bit more straightforward than coding

• Start at the root of the tree, and follow links to “0” or “1” children, depending on the next bit in the code

• When you reach a leaf, the symbol you’ve just decoded is found in it
Decoding path traversals in a Huffman code tree

• Exactly the same tree (or equivalent information) needs to be used for decoding and coding, to ensure that you get out of the code what you put into it

• To decode, you follow a path from the root of the tree to a leaf
  • this requires that each internal node have links to its “0” and its “1” child

• You output the value of the coded symbol when you reach a leaf
  • this requires that each leaf node contain or point to the value of the corresponding symbol
Nodes in a Huffman code tree

• So we have found that to be useful for both coding and decoding, a node in a Huffman code tree should have:
  • a pointer to its parent (which will be null, if it is the root of the tree)
  • some way to tell whether this is the “0” or “1” child of its parent
  • a pointer to its “0” child and its “1” child (which will both be null, if it is a leaf)
  • the value of the alphabet symbol (if it is a leaf)

• You also should have a fast way, given a symbol to code, to get to the code tree leaf containing that symbol

• There are various ways to implement these requirements...
Ways to implement a tree

• As you know, in computer science, if there’s one way to do something, there’s infinitely many ways to do it

• When implementing a tree, for example, you can
  • use dynamic data and pointers
    • flexible, general purpose. Clearly the best choice when you don’t know the structure or the size of the tree beforehand
  • use an array
    • fast, compact. Usually the best choice when implementing heaps, or other trees with a very regular structure

• You can use either of these to implement a Huffman code tree.
• We will consider both of them, and also look at other issues that can affect the time and space costs of Huffman’s algorithm, and of using the resulting tree to code and decode messages
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)
Huffman’s algorithm

1. Determine the count of each symbol in the input message.
2. Create a forest of single-node trees. Each node in the initial forest contains a symbol from the set of possible symbols, together with the count of that symbol in the message to be coded. Symbols with a count of zero are ignored (consider them to be impossible).
3. Loop while there is more than 1 tree in the forest:
   a) Remove the two trees from the forest that have the lowest count contained in their roots.
   b) Create a new node that will be the root of a new tree. This new tree will have those two trees just removed in step 3a as left and right subtrees. The count in the root of this new tree will be the sum of the counts in the roots of its subtrees. Label the edge from this new root to its left subtree “1”, and label the edge to its right subtree “0”.
   c) Insert this new tree in the forest, and go to 3.
4. Return the one tree in the forest as the Huffman code tree.
Time cost analysis of Huffman’s algorithm

• Suppose the input message is a sequence of length K drawn from an alphabet of N symbols
  • These two parameters define the size of a Huffman coding problem... so we will define space and time costs in terms of them

Step 1: time cost $O(K)$
Step 2: time cost depends on the data structure used to hold the forest
Step 3: the loop is executed ______ times
Step 3a: time cost depends on the data structure used to hold the forest
Step 3b: time cost $O(1)$
Step 3c: time cost depends on the data structure used to hold the forest
Step 4: time cost $O(1)$

• Overall time cost will depend on the performance of the data structure used to hold the forest, so let’s look at that in detail
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
Queues and Priority Queues

- An (ordinary) Queue is an abstract data type with these operations:
  - Create an empty Queue
  - Insert an item into the Queue
  - Delete, and return, an item in the Queue

- Of all the items currently in a Queue, the Delete operation must return which one?.....
  (So, a Queue is often called a ____________ structure.)

- A Priority Queue is an abstract data type with these operations:
  - Create an empty Priority Queue
  - Insert an item into the Priority Queue
  - Delete, and return, an item in the Priority Queue

- The difference is: in a Priority Queue, the Delete operation must return the item with the highest priority
- So, items stored in a Priority Queue must be comparable to each other with respect to their “priority”
Implementing a priority queue using a linked list

- One common way to implement a queue is with a linked list
  - Maintain pointers to the beginning of the list, and the end of the list
  - Insert items at the end; delete items from the beginning (or vice versa)

- A linked list could also be used to implement a priority queue
  - Maintain the list in sorted order, with the “highest priority” item first

- Inserting an item in such a priority queue requires traversing the list from front to back to find the place to insert the item, in order to keep the list sorted
  - with N items has average- and worst-case time cost $O(\_\_\_\_)$

- Deleting the highest priority item in such a priority queue just requires unlinking the first item in the list
  - with N items has best-, average-, and worst-case time cost $O(\_\_\_\_)$
Analysis of Huffman’s algorithm with linked-list 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: $O(N^2)$, or $O(N \log N)$ if counts are sorted first
Step 3: the loop is executed N-1 times
  - Step 3a: time cost $O(1)$
  - Step 3b: time cost $O(1)$
  - Step 3c: time cost $O(N)$
Step 4: time cost $O(1)$

- So overall time cost: $O(K + N^2)$
- You can do better using a heap, instead of a linked list, to implement the priority queue
Next time...

- Heaps and Priority Queues
- Using `std::priority_queue`
- Time and space costs of coding with Huffman codes
- Dynamic data and array representations for Huffman trees