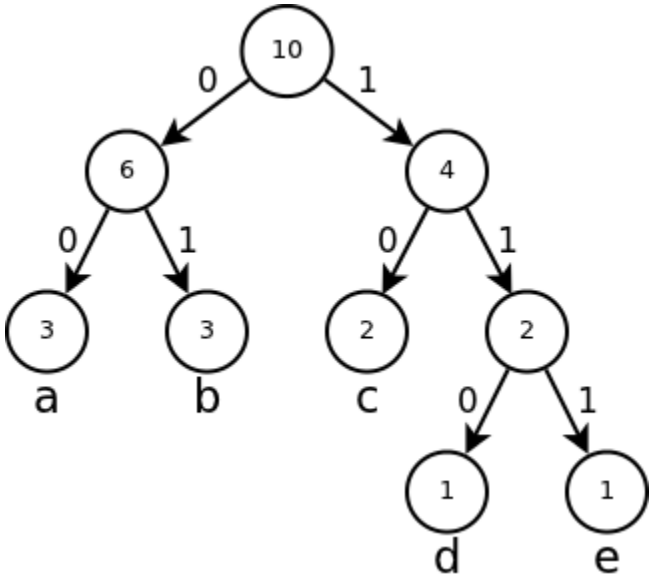
ADS PROJECT

Himanshu Taneja(1244-9584)

APRIL 5, 2017

HUFFMAN CODING



Structure Of Program

The files are separated as .cpp and .h files.

.cpp: files have implementation methods

.h header files: contain classes declarations and function prototypes.

1. **binary**: Heaps are used for the implementation of the priority queue. The Huffman coding needs two minimum elements to be extracted and then the sum of those extracted elements to be inserted in the heap. Since min heaps need to maintain their properties, they use functions like heapify up and heapify down.

2. **fourway**: Achieves cache optimization by moving the elements to start from 4th postion. (index/4)+2 to find the parent of that index. (4 * (index-2) + (k-1)) as kth child of node i. Since min heaps need to maintain their properties, they use functions like heapify up and heapify down after any update to the tree like delete or insert.

3. **pairing**: The pairing heap is different than the previous two. The insert function inserts the element in the tree but on deleting the minimum node from the tree, the subtrees are combined using combineSiblings. The compareAndLink funcions compares two subtrees and the subtree with greater root node is attached to the other subtree.

FUNCTION PROTOTYPES:

hnode.h:

It is defined as a class binary tree node which is attributed with data, frequency value, and some designated pointers as left and right to efficiently parse through the nodes of our generated Huffman Trees.

Encoder.cpp:

1. void FillFreqMapFromFile-
   Creates Frequency Table.

2. hnode const* BuildTreeFheap(std::unordered_map<int,unsigned int> const &freq_map)-
   This function builds Huffman tree by utilizing a 4-way cache optimized heap

3. hnode const* BuildTreeBheap(std::unordered_map<int,unsigned int> const &freq_map)-
   Utilizing the Binary heap

4. hnode const* BuildTreePheap(std::unordered_map<int,unsigned int> const &freq_map)-
   This function builds Huffman tree by utilizing a Pairing heap

5. void CreateHnodeVecFromFreqMap(std::unordered_map<int,unsigned int> const &freq_map, std::vector<hnode const*> &vec)-
   Creates List of Huffman nodes

6. void printCodes(hnode const *root, std::string str,
   std::unordered_map<int,std::string> &code_table_map)-
   This function traverses through Huffman tree to generate key value pairs for generating code table.

7. void compress(std::string &value,std::vector<char> &result)-
   It compresses the string by using 8-bit compression process.

8. void binfilecreate(std::unordered_map<int,std::string> &code_table_map, char *filename, char const *out)-
   This function creates final encoded binary by using hash map and code table generated intermediately.

Decoder.cpp:

1. struct Node {struct Node *right = NULL; struct Node *left = NULL; int value; bool isLeaf = false;};- Defining the node of the Huffman tree which is intended to be decoded.

2. Node* getRoot()- Gets the root of the Huffman tree.

3. void addLeafAt(int value, string binaryString)-
   Constructs the decode-huffman tree after parsing code_table.txt

4. int getAt(string binaryString)-
   Traverses through decode tree to parse the binary string and returns valid value stored in leaf node.

5. The primary function of decoding the entire encoded.bin file is executed by the main of decoderHuffman.cpp.

PERFORMANCE ANALYSIS

The average time required to create Huffman Tree 10 times for all three heaps is measured as :

- For **binary heap**, the time required was 2.17 seconds
- For **four way heap**, the time required was 1.75 seconds
- For **pairing heap,** the time required was 2.19 seconds

Since the Four way heap is the fastest, the Huffman coding was done using data structure fourway heap,

The time required for the encoding is 31 seconds while for decoding the time required is 28 decoding.

Explanation:

Huffman Tree is generated using all three data structures and four way heap turns out to the most efficient.

The four way is fastest because

- It has smaller height compared to the other two. When the input is very large, the height of the tree matters.
- Cache optimization is more in Four-way because the index of the parents starts from 3, the children will thus be at indices 4, 5, 6, 7. Thus the siblings will be at consecutive indices i.e. same cache line as shaded below.

DECODING ALGORITHM AND COMPLEXITY

1) Open the files encoded.bin and code_table.txt
2) Consider the code table which has the leaf node value and its code separated by a space and construct a Huffman Tree.
   a) Using the code, create nodes in the tree, if 0 then create a new node as left child else if 1 then create a new node as right child.
   b) Once the code length is over, insert the value to that node and make it a leaf node.
   c) Repeat till all the Tree is constructed.
3) Then traverse through Huffman tree using binary string till you reach the leaf and return the leaf node value.
   a) If binary string is at 0, move to the left child else if 1, then move to the right child of the current node.
   b) If the node is a leaf node, it will have the value stored in it, which will be returned.
4) Write the leaf node value in the file called decoded.txt

The complexity of the code will be height times number of nodes.

⇨ O(n.h) time