Advanced Data Structures (COP 5536)

Spring 2017

Programming Project Report

Abhineet Rajendra Kulkarni

UFID – 13810085

abhineetrkulkarn@ufll.edu

Project Description

Basic Idea of this project is to implement Huffman Coding and use three different data structures to implement the priority queue. Select the best of the three and use it to implement encoder and decoder.

The structures used are:

- Binary Heap
- Four-Way Cache Optimised Heap
- Pairing Heap

The best time for construction of Huffman Tree resulted from Binary Heap which is used for encoder and decoder. The detailed performance analysis is given in the following pages. The program structure and algorithm used for the decoder also follow.

Programming Environment

The project has been implemented using Java. Code has been tested on Java environment of thunder.cise.ufl.edu and it produces the results as expected. Makefile is used to produce executable files encoder and decoder according to project requirement.

The file encoder will take one argument as the name of the input file. It will produce two outputs:

- encoded.bin – The encoded binary file of the input.
- code_table.txt – The code table listing the Huffman code of each unique input.

The file decoder will take two files as input and produce one output. The input files will be the name of the encoded file and the name of the code table file. The output produced will be:

- decoded.txt – The exact same file as the input file passed to encoder

Program Structure

The program is divided into 3 main parts, namely the Test Structures for Huffman Coding, Encoder and Decoder. The classes are also divided accordingly and written below. The part of Huffman Coding focusses on implementing the priority queue using different data structures and analysing their performance and selecting the best possible one.

Binary Heap:

Packages Imported:-

- import java.util.ArrayList;
- import java.util.List;
- import java.util.Map;
- import java.util.Scanner;
- import java.util.HashMap;
- import java.io.FileReader;
- import java.io.BufferedReader;
- import java.io.FileNotFoundException;
- import java.io.IOException;
- import java.io.PrintWriter;

Classes present in Binary Heap:

1. node
2. binary_heap

Node:

- This class contains the basic structure of the node used for binary heap and the get/set functions for its properties.
- Properties of Node:-
  - I. leftChild – Pointer to the left child.
  - II. rightChild – Pointer to the right child.
  - III. freq – The frequency of the data.
  - IV. data – The data field.

- Methods supported in this class:
  - **public** Integer fetchData():
    - Used to get the data present in the node.
  - **public void** setData(Integer d):
    - Used to set the data of the node to a particular value.

- **public** Integer getFreq():
    - Used to get the frequency of the data at that node.
- **public void** setFreq(Integer f):
    - Used to set the frequency of the node to a particular value.
- **public** node getlc():
    - Used to get the left pointer of the node.
- **public** node getrc():
    - Used to get the right pointer of the node.
- **public** node setlc(node l):
    - Used to set the left pointer of the node.
- **public** node setrc(node r):
    - Used to set the right pointer of the node.

Binary Heap:

- This class is used for the implementation of the priority queue using binary heap. It uses the node structure described above.
- The properties of this class are:
    I. heapArr: An ArrayList of type node to store all the nodes.
    II. inputFilePath: The path of the input sample file.
    III. Frequency_Table: A HashMap used to store the frequency tavle.
    IV. startTime: Used to measure time required to build the heap.
- Methods supported in this class:
    - **public void** insert(node x):
        - Used to insert a node into the binary heap
    - **public void** buildHeap(String inputFilePath):
        - This method is used to build the frequency table from the input file and insert is called to insert the nodes into the binary heap for each entry in the frequency table.
    - **private void** shiftup(**int** n):
        - This method is used to check the heap property on a newly inserted node.
    - **public** node extractMin():
        - This method is used to extract the smallest element from the heap. The last element from the leftmost subtree replaces the root.
    - **private void** shiftdown(**int** n):
        - This method is used to shift the newly replaced root to its correct position and restore heap property.
    - **private int** minIndex(**int** n):
        - Used to get the index of the smallest child.
    - **public int** getSize():
        - Used to get size of the priority queue.

- **private int** parent(**int** n):
  - Used to get the index of the parent of a node at index n.
- **private int** left(**int** n):
  - Used to get the index of the left child of a node at index n.
- **private int** right(**int** n):
  - Used to get the index of the right child of a node at index n.
- **private void** swapnodes (**int** index1, **int** index2):
  - Used to swap 2 nodes by comparing their frequencies. Invoked by shiftdown during traversal of the tree.
- **public** node createHufftree():
  - Used to create the Huffman Tree. Calls insert to insert nodes and createnode to create Huffman nodes. Returns root of the Huffman Tree.
- **public** node createnode(node nodeOne, node nodeTwo):
  - Takes 2 nodes and sets the frequency of the parent as the addition of the individual frequencies. Sets its left and right children and return the new node.
- **public void** createHuffcodes(node root, String s, FileWriter w):
  - Takes the root of the Huffman Tree and assigns codes according to the algorithm. The codes are written to the file by the Fileriter.

Four-Way Cache Optimised Heap:

Packages Imported:-

- import java.util.ArrayList;
- import java.util.List;
- import java.util.Map;
- import java.util.Scanner;
- import java.util.HashMap;
- import java.io.IOException;
- import java.io.FileWriter;
- import java.io.File;

Classes present in Binary Heap:

1. fourway_heap_node
2. fourway_heap

fourway_heap_node:

- Fourway heap uses a similar structure to node class. The only difference being that it has four children. The basic structure of the class is similar to node and it just uses an object of node to access all of the methods defined in class node.
- Properties of this class:
  I. Object of node n: Used to access methods defined in class node.
- Methods defined in this class:
  - **public** Integer getData():
    - Used to get data from node.
  - **public void** setData(Integer d):
    - Used to set the data of a node.
  - **public** Integer getFreq():
    - Used to get the frequency of a data item.
  - **public void** setFreq(Integer f):
    - Used to set the frequency of a node.
  - **public** node getnode():
    - Used to return the object node.
  - **public void** setlc(fourway_heap_node l):
    - Used to set l as the logical left child of the node n.
  - **public void** setrc(fourway_heap_node r):
    - Used to set r as the logical right child of node n.
  - **public** node getlc():
    - Returns the left pointer of the node.
  - **public** node getrc():
    - Returns the right pointer of the node.

fourway_heap:

- This class is used for implementing thee priority queue for generating the Huffman Tree using fourway cache optimized heap.
- Its properties are:
  I. **private** List<fourway_heap_node> fwheap: An ArrayList of type fourway_heap_node to store all the elements.
  II. **private** HashMap<Integer, Integer> Frequency_Table: A HashMap for storing the frequency table generated from the input.
  III. **private** String path: Used to store the path of the input file.
  IV. **public long** starttime: Used to measure the time.

- The Methods in this class are:
  - **public void** build():
    - Used to build the frequency table and the heap.
  - **public void** insertintoheap(fourway_heap_node num):
    - Used to insert a new node into the fourway_heap. Called in build().
  - **public void** minheap(Integer index):
    - Used to restore the min heap property after each insert or extractMin().
  - **public void** swap(Integer node1, Integer node2):
    - Used to swap two nodes if they violate the min heap property. Called in minheap.
  - **public** Integer getminchildindex(Integer numindex):
    - Used to get the index of the minimum child from all the four children. Called by minheap before calling swap().
  - **public** fourway_heap_node extractMin():
    - Returns the root of the tree, sets the last element to be the new root and calls minheap().
  - **public** node createhufftree():
    - Used to create the Huffman Tree by removing two smallest elements from the queue and passing them to createNode to make a new internal node and then calling insert to insert it back to the queue. Returns the root of the tree.
  - **public** fourway_heap_node createNode(fourway_heap_node N1, fourway_heap_node N2):
    - Creates a new Huffman Node and sets its children and returns it back to createhufftree().
  - **public void** createhuffcodes(node root, String code, FileWriter w):
    - Takes in the root of the Huffman Tree, a blank string and a fileWriter to write the Huffman Codes to a file. Generates codes by appending zeroes or ones to the previous code as we traverse the tree.
  - **public** Integer getc1Index(Integer n):
    - Returns the index of the first child of the node n which it takes as an argument.
  - **public** Integer getc2Index(Integer n):
    - Returns the index of the second child of the node n which it takes as an argument.
  - **public** Integer getc3Index(Integer n):
    - Returns the index of the third child of the node n which it takes as an argument.
  - **public** Integer getc4Index(Integer n):
    - Returns the index of the fourth child of the node n which it takes as an argument.

Pairing Heap:

Packages Imported:

- import java.util.ArrayList;
- import java.util.List;
- import java.util.Map;
- import java.util.Scanner;
- import java.util.HashMap;
- import java.io.IOException;
- import java.io.FileWriter;
- import java.io.File;

Classes present in Pairing Heap:

1. node
2. binary_heap

Pairing_Heap_Node:

- This class contains the basic structure of the node used for pairing heap and the get/set functions for its properties.
- Properties are:
    I. Node: An object used to access the original node class.
    II. childptr: Used to access the child.
    III. leftptr: Used to access the next item in the linked list.
    IV. rightptr: Used to access the previous item in the linked list.
- The methods in this class are:
    - **public** Integer getData():
        - Used to get data from node.
    - **public void** setData(Integer d):
        - Used to set the data of a node.
    - **public** Integer getFreq():
        - Used to get the frequency of a data item.
    - **public void** setFreq(Integer f):
        - Used to set the frequency of a node.
    - **public** pairing_heap_node getleftptr():
        - Used to get the left pointer of a node.
    - **public** pairing_heap_node getrightptr():
        - Used to get the right pointer of a node.
    - **public void** setleftptr(pairing_heap_node l):
        - Used to set the left pointer of a node.
    - **public void** setrightptr(pairing_heap_node l):

    - Used to set the right pointer of a node.
- **public** pairing_heap_node getchildptr():
    - Used to get the child pointer of a node.
- **public** pairing_heap_node setchildptr():
    - Used to set the child pointer of a node.
- **public void** setlc(pairing_heap_node l):
    - Used to set l as the logical left child of the node n.
- **public void** setrc(pairing_heap_node r):
    - Used to set r as the logical right child of node n.
- **public** node getlc():
    - Returns the left pointer of the node.
- **public** node getrc():
    - Returns the right pointer of the node.

Pairing_Heap:

- This class is used for implementing thee priority queue for generating the Huffman Tree using pairing heap.
- Its properties are:
    I. **private** List<fpairing_heap_node> fwheap: An ArrayList of type fourway_heap_node to store all the elements.
    II. **private** HashMap<Integer, Integer> Frequency_Table: A HashMap for storing the frequency table generated from the input.
    III. **private** String path: Used to store the path of the input file.
    IV. **public long** starttime: Used to measure the time.
- The methods in this class are:
    - **public void** build():
        - Used to build the frequency table and the heap.
    - **public void** insertintoheap(pairing_heap_node n1):
        - Used to insert a new node into the fourway_heap. Called in build().
    - **public** pairing_heap_node getroot():
        - Returns the root of the tree.
    - **public** pairing_heap_node removemin():
        - Used to remove the root of the tree and calls meld to establish the min heap property.
    - **public** pairing_heap_node Meld(pairing_heap_node n):
        - Called by removemin() to establish the min heap property. Reconfigures the pointers to make it a min heap.
    - **public** node createhufftree():
        - Used to create the Huffman Tree by removing two smallest elements from the queue and passing them to createNode to make a new

    - internal node and then calling insert to insert it back to the queue. Returns the root of the tree.
- **public** pairing_heap_node createNode(pairing_heap_node N1, pairing_heap_node N2):
    - Creates a new Huffman Node and sets its children and returns it back to createhufftree().
- **public void** createhuffcodes(node root, String code, FileWriter w):
    - Takes in the root of the Huffman Tree, a blank string and a fileWriter to write the Huffman Codes to a file. Generates codes by appending zeroes or ones to the previous code as we traverse the tree.

Conclusions:

It was observed that Binary Heap was the fastest among all three data structures. Four Way Cache Optimised Heap was slowed down due to the fact that it was not properly optimised for the test environment. I used the multipass melding scheme for Pairing Heaps which might have caused it to slow down. Binary Heap being the fastest is used as the priority queue for encoding. The times required for generating the Huffman Codes using the three different structures are given below:

- Binary Heap – 2179 ms
- Four Way Cache Optimised Heap – 2654 ms
- Pairing Heap – 233406 ms

Encoder

Packages Imported:

- **import** java.io.BufferedOutputStream;
- **import** java.io.BufferedReader;
- **import** java.io.FileOutputStream;
- **import** java.io.FileReader;
- **import** java.io.FileWriter;
- **import** java.io.IOException;
- **import** java.io.OutputStream;
- **import** java.util.HashMap;
- **import** java.util.Map;

Encoder has the following methods:

- **private static** HashMap <Integer, Integer> buildFreqTable(String inputFilePath):
  - Builds the frequency table from the input file. This method takes the path of the input file as an argument and returns the HashMap with the values.
- **private static** node createHuffTree(binary_heap bh):
  - Takes as input an object of binary_heap and calls the insert function from binary_heap.java to create a new Huffman node and insert it in the priority queue. Returns the root of the Huffman Tree.
- **private static void** createHuffCode(node root, String c, HashMap<Integer, String> codeTable, FileWriter codeTableFile):
  - Takes as input the root of the Huffman Tree and an empty string alongwith an empty HashMap and a FileWriter. Generates the file code_table.txt as output and writes all the codes to the HashMap.
- **public static void** encode():
  - Calls the functions createHufftree and createHuffCode and generates the encoded.bin file as output. Appends all the codes to encodedFS and converts them to binary to write to encoded.bin

Decoder

Packages Imported:

- **import** java.io.BufferedReader;
- **import** java.io.File;
- **import** java.io.FileReader;
- **import** java.io.IOException;
- **import** java.io.PrintWriter;
- **import** java.nio.file.Files;

Decoder has the following methods:

- **private static** node createHuffTree(String codeTablePath): Takes as argument the path of code_table.txt and constructs the Huffman Tree from it.
- **public static** String getEncodedString(String encodedPath): Takes as argument the path of encoded.bin and reads the entire file to a string.
- **private static void** decode(): Calls both createHuffTree() and getEncodedString() and using both constructs the original sample input file and writes that to decoded.txt

Decoding Algorithm:

- First step involves reading the file code_table.txt and construct the decode tree by traversing down the tree following the code, going to the left if it's a zero or the right if it's a one. At the place you fall of the tree create a new node, and assign the corresponding data from code_table.txt to that node. The complexity of this will be the sum of the codes of individual lengths. If the code lengths of n nodes are $c_{1,}$ $c_{2,}$ …. ,… , c_n the complexity will be,

  Time for generating Decode Tree = O$(c_{1} + c_{2} + \ldots + c_{n})$

- Next step involves reading all the bytes from encoded.bin to a string.
- Now, we traverse the decode tree bit by bit following the encodedFileString till we fall off the tree. Once we fall off, we write the data to decoded.txt. Then, we check the next bit and depending on whether it's a zero or one we traverse to the left or right child of the root.
- We do this for the entire length of the encodedFileString.

Time complexity of the Decoding algorithm = Time for generating decode tree + Time for generating decoded.txt

= $O(c_{1} + c_{2} + \ldots + c_{n})$ + *O(encodedFileString.length())*

Conclusions:

Huffman codes provides a great way for transmission of data. The size of the encoded file was less than 1/3rd of the original sample data. A Binary Heap based priority queue gave the best performance among all the provided data structure options.