Haskell Interim Test

Imperial College London

Department of Computing

Data Compression

Friday 07 November 2025

14:00 to 16:00

TWO HOURS

(including 10 minutes planning time)

- The maximum total is **25 marks**.
- Credit is awarded throughout for conciseness, clarity, and the appropriate use of the various language features and concepts.
- **Important: 10%** is deducted from solutions that do not compile on lab machines. Comment out any code that does not compile before you finish, *leaving stubs (`f = undefined`) behind if necessary, the tests must compile too!*
- The examples and test cases here are not guaranteed to exercise all aspects of your code. You are advised (but not required) to define your own tests to complement the ones provided.
- The files are in your Home folder, under the "`compress`" subdirectory. **Do not move any files**, or the test engine will fail resulting in a compilation penalty.
- When you are finished, simply **save everything and log out**. **Do not shut down your machine**. We will fetch your files from the local storage.

NOTE: `Data.List` and `Data.Map`

The skeleton files import the `Data.List` and `Data.Map` modules. These contain various useful functions for you to use. `Data.Map` has been imported *qualified* as `Map`, so you must prefix any functions you use with `Map.`. You may use functions from `Data.List` without qualification.

You can use the `cabal repl`/`ghci` commands: `:browse` and `:doc` to find relevant functions for you to use and see their documentation, respectively. For example:

```
ghci> :browse Data.List
isSubsequenceOf :: Eq a => [a] -> [a] -> Bool
(!!) :: GHC.Stack.Types.HasCallStack => [a] -> Int -> a
(++) :: [a] -> [a] -> [a]
(\\) :: Eq a => [a] -> [a] -> [a]
all :: Foldable t => (a -> Bool) -> t a -> Bool
[...]

-- (with Data.List imported)
ghci> :doc sort
-- Identifier defined in `base-4.18.3.0:Data.OldList'
-- | The 'sort' function implements a stable sorting algorithm.
-- It is a special case of 'sortBy', which allows the programmer to supply
-- their own comparison function.
--
-- Elements are arranged from lowest to highest, keeping duplicates in
-- the order they appeared in the input.
--
-- >>> sort [1,6,4,3,2,5]
-- [1,2,3,4,5,6]
--
-- The argument must be finite.
--
```

Data Compression using Huffman Coding

The standard ASCII code (a subset of Unicode) uses 7 bits to encode each character, so, for example, a string with 5 characters requires 35 bits in total. Data compression is all about representing the same information using fewer bits. As an example, one way to compress English text is to exploit the fact that some letters in the language are much more common than others[1]. The idea is to use a binary tree – a so-called *Huffman coding tree* – and to arrange for the most commonly-occurring letters to appear close to the root of the tree. A character is then encoded by a path through the tree: the more common the character, the shorter the path.

Note that we can build a Huffman coding tree from a list of any type `a`, so long as we can compare values of the type `a` for `Eq`uality. However, for this test, we will strengthen that constraint to be for *`Ord`ered* types instead: this (theoretically) allows us to construct the tree in $O(n \log n)$, compared to $O(n^2)$ when using an `Eq` constraint, which is decidedly more efficient.

As an example, fig. 1 shows the tree for the string "mississippi is missing", i.e. where the ordered type here is `Char`. Using this tree the character `'m'` would be encoded by the path `011`,

[1]It turns out that the most common letter in English is 'e', followed by 't', followed by 'a', and so on.

where `0` and `1` respectively mean “go left” and “go right” through the tree. A list of characters can then be encoded by concatenating the paths corresponding to each character. For example, using the same tree, the string “sips” would be coded as: `111000011`; this uses just 9 bits (binary digits) as opposed to 28 bits for standard ASCII.

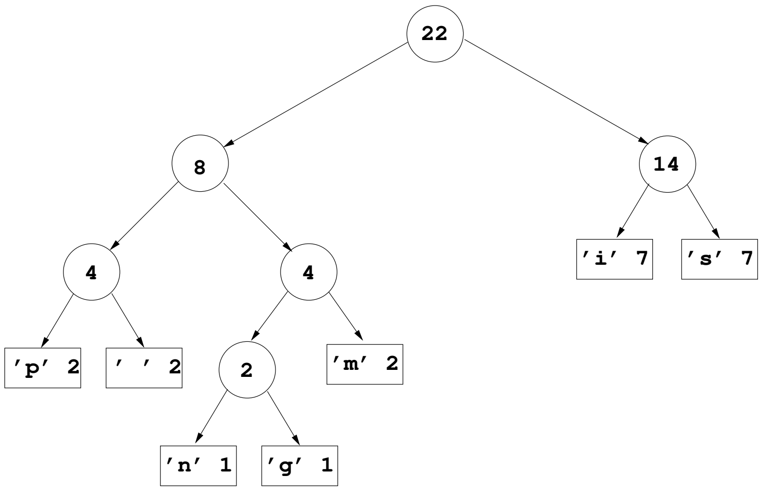

Figure 1: The Huffman coding tree for “mississippi is missing”

When building a Huffman tree we need to keep track of the *frequency counts* (the number of times a letter appears in a string) of each item in the list used to build it. Each leaf (`Leaf`) will store both an item from the source list and a count of the number of times that item appeared. Each internal node (`Node`) will store, in addition to its two subtrees, the sum of the frequency counts of all the `Leaf` items in those subtrees. In this sense the tree is *well formed* and all trees in this exercise can be assumed to be well formed. Figure 1 shows the frequency counts in each node, including the leaf nodes; there is precisely one node for each unique character in the input string. For example, the tree tells us that the original string had two ‘p’s, two ‘m’s, one ‘g’, seven ‘i’s and so on. The most common characters were ‘i’ and ‘s’, which is why they appear nearer the top of the tree than, say, ‘g’.

Huffman trees with these added counts can be represented in Haskell by the polymorphic data type `HTree`:

```haskell
data HTree a = Leaf a !Int | Node !Int (HTree a) (HTree a) deriving (Show)
```

Note that this allows trees to contain any type – the items do not necessarily have to be `Char`s.

One tree is defined to be ‘equal’ to another if their frequency counts are the same. Similarly, one tree is defined to be ‘smaller’ than another if its frequency count is smaller. To implement this, you have been given `Eq HTree` and `Ord HTree` instances, so that the operators `(==)`, `(/=)`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max` and `min` are all defined appropriately on `HTree`s. Thus, for example,

```
ghci> Leaf 'c' 3 <= Leaf 'd' 1
False
ghci> max (Leaf 'a' 4) (Node 12 (Leaf 'x' 8) (Leaf 'y' 4))
Node 12 (Leaf 'x' 8) (Leaf 'y' 4)
```

Make sure you understand this before reading on.

What to do

Completing `Data.HTree` **worth 12% of the test**

The file `src/Data/HTree.hs` contains part of an implementation of a `HTree` with two incomplete functions: `freqCount` and `merge`.

1. Implement the function `freqCount :: HTree a -> Int`, which returns the total frequency count of all leaves in the given tree.

   Hint: pay attention to the invariances listed for `HTree`.

2. Implement the function `merge :: HTree a -> HTree a -> HTree a`, a so-called *smart constructor* for `HTree` that helps ensure the invariances are always respected when constructing a new `Node`.

   To merge two trees `t1` and `t2` with frequency counts `n1` and `n2` respectively you simply build a new `Node` whose frequency count is `n1+n2`, whose left subtree is the 'smaller' of `t1` and `t2`, and whose right subtree is the 'larger' of `t1` and `t2`.

   Hint: this is a simple non-recursive function.

Completing `Data.Bag` **worth 16% of the test**

A *bag* (or multi-set) is a data structure that acts as a *set* allowing duplicates. A `Bag a` can be simply represented as a wrapper around `Map a Int`, where the `Int` is the *multiplicity* (the number of occurrences) of an element in the set. For any item x in a bag, its multiplicity in the underlying map is guaranteed to be ≥ 1. The file `src/Data/Bag.hs` contains part of an implementation of a `Bag`: you are given the `Bag` type, as well as `empty` and `toList` for bags.

3. Define the function `insert :: Ord a => a -> Bag a -> Bag a`, which given a value `x`, inserts it into the given bag. If the element was not already found in the bag, its multiplicity should now be 1. Alternatively, if it already existed in the bag with multiplicity n, the multiplicity should be changed to $n + 1$. For example:

   ```
   ghci> Bag.empty
   fromList []
   ghci> Bag.insert '5' (Bag.insert '5' Bag.empty)
   fromList [('5',2)]
   ghci> Bag.insert '4' (Bag.insert '3' Bag.empty)
   fromList [('3',1),('4',1)]
   ```

   Hint: consider relevant functions in `Data.Map`.

4. Define the function `fromList :: Ord a => [a] -> Bag a`, which puts all the elements of the given list into an empty bag and returns it. For example:

   ```
   ghci> Bag.fromList [True, True, False, False, True, False, True]
   fromList [(False,3),(True,4)]
   ghci> Bag.fromList "hello world"
   fromList [(' ',1),('d',1),('e',1),('h',1),('l',3),('o',2),('r',1),('w',1)]
   ```

Building a Huffman Tree **worth 32% of the test**

You will now work in `src/Compress.hs`. You're now going to build a (Huffman) tree from a given source list, `xs`. To do this you start by using `occurrences` to build a count of the number of times each element occurs in `xs`. Now, for each element `(x,n)` in the result, build the tree `Leaf x n`. You now have a list of trees, all of which are leaves. Now use the `sort` function from `Data.List` to sort the trees into *ascending* order of their frequency counts. Note that `sort` will do the right thing here because of the defined ordering on trees in terms of the frequency counts (see above). As an example, given the string `"mississippi is missing"` (this string is predefined and called `testString` in `src/Examples.hs`), the `occurrences` function first generates the list `[('m',2), ('i',7), ('s',7), ('p',2), (' ',2), ('n',1), ('g',1)]`. After turning each element into a leaf and sorting the list of leaves you get: `[Leaf 'n' 1, Leaf 'g' 1, Leaf 'm' 2, Leaf 'p' 2, Leaf ' ' 2, Leaf 'i' 7, Leaf 's' 7]`.

Now you do the following to the above list of trees: take the two smallest (i.e. leftmost) trees in the list, *merge* them to form a bigger tree, and then use the `insert` function from the `Data.List` module to insert the merged tree into the remaining elements of the list, i.e. preserving the frequency count order. Observe that the new list now contains one fewer elements than the original. Now repeat the process until the list contains just a single tree – this tree is the required result.

As an example, starting with the above list we first combine the two leftmost trees giving `Node 2 (Leaf 'n' 1) (Leaf 'g' 1)`. We then insert this tree into the remainder of the original list giving the new list `[Node 2 (Leaf 'n' 1) (Leaf 'g' 1), Leaf 'm' 2, Leaf 'p' 2, Leaf ' ' 2, Leaf 'i' 7, Leaf 's' 7]`. Continuing the process we eventually end up with a *singleton* list comprising just the tree shown in fig. 1 – this is called `fig` in `src/Examples.hs`.

5. Use `Bag` to define the function `occurrences :: Ord a => [a] -> [(a, Int)]` that given a list `xs` will return, for each element, `x`, the pair `(x,n)`, where `n` is the number of times `x` occurs in `xs`. For example, `occurrences "mississippi"` should return `[('m',1), ('i',4), ('s',4), ('p',2)]`, in any order. Note that `Data.Bag` has been imported qualified as `Bag`.

6. Define the function `reduce :: [HTree a] -> HTree a` that given a **non-empty** and **sorted** list of trees will return a single tree by repeated application of `merge` and `insert`, as above.

   Hint: You can use pattern matching to pick off the first two elements of a given list:

   ```
   reduce (t1 : t2 : ts) = ...
   ```

7. Define the function `buildTree :: Ord a => [a] -> HTree a`, which combines together `occurrences`, `sort`, and `reduce` to generate the tree corresponding to the given list. A precondition is that the list is non-empty. For example `buildTree "mississippi is missing"` should generate the `HTree` in fig. 1, i.e. `buildTree testString == fig` should return `True`.

The test continues on the next page

Performing Compression **worth 28% of the test**

Your next task is to define the `encode` and `decode` functions: if you get stuck with one, move onto the other, which you may find easier.

8. Define a function `encode :: Eq a => HTree a -> [a] -> Code`, which takes a Huffman tree along with a list of items to encode and produce a list of 0s and 1s (type `Code`) representing the bitstring Huffman encoding of the list of items. For example, `encode fig "sign"` should return `[1,1,1,0,0,1,0,1,0,1,0,0]`. A precondition is that the given tree can encode each of the items in the list, i.e. each item has **exactly one** corresponding leaf somewhere in the tree. Ideally, this function only traverses the tree once per item in the list: carefully consider efficiency and elegance.

   Hint: items in the list are independently encoded. For an item, consider using a helper function with an accumulating parameter to build paths from the root of the `HTree` *to each leaf. Notice that at any forks in the tree at most one of the paths would be needed... can you think of a way to quickly/cheaply distinguish the path to the correct leaf from useless ones?*

9. Define a function `decode :: HTree a -> Code -> [a]`, which takes a Huffman tree and an encoded list of items and returns the corresponding (decoded) list. For example, `decode fig [1,1,1,0,0,0,0]` should return `"sip"`. Note that, in general, if tree `t` can encode list `xs` then `(decode t . encode t) xs` should return `xs`. A precondition is that the code is valid with respect to the tree (i.e. that every complete group of bits in the code describes a valid path into the tree). This function should do work proportional to the number of bits in the code ($O(n)$).

Encoding the tree **worth 12% of the test**

These two functions are low weighted yet difficult, so **you should only spend time on them when you have completed the above**.

To use a Huffman tree for data compression you need to be able to compress the tree as well as the data you are trying to encode; otherwise you can't decode the data! We can define a compression algorithm for the tree with the following scheme: a `Leaf` is encoded as a 1 followed by 7 bits representing the ordinal value of the character at the leaf , itself encoded as a bitstring (the most-significant bit should be at the front, so that `'a'` is `[1,1,0,0,0,0,1]`); a `Node` is encoded as a 0 followed by the encoding of the left subtree followed by the encoding of the right subtree.

10. Define a function `compressTree :: HTree Char -> [Int]` that will compress a tree of characters into a bitstring (here a list of 0s and 1s) using the above scheme. For example:

    ```
    ghci> compressTree fig
    [0,0,0,1,1,1,1,0,0,0,0,1,0,1,0,0,0,0,0,0,0,1,1,1,0,1,1,1,0,1,1,
     1,0,0,1,1,1,1,1,1,0,1,1,0,1,0,1,1,1,0,1,0,0,1,1,1,1,1,0,0,1,1]
    ```

    It is possible, though more challenging, to write this function to perform a fixed amount of work for each `Leaf`/`Node` in the tree ($O(n)$): you may wish to try this, but make sure the function is still clean.

11. Define a function `rebuildTree :: [Int] -> HTree Char` that will build a Huffman tree of characters from its bitstring encoding. You only need the frequency counts during

the initial construction of a tree so you should just set these to 0 when reconstructing. For example:

```
ghci> buildTree "suss"
Node 4 (Leaf 'u' 1) (Leaf 's' 3)
ghci> compressTree (buildTree "suss")
[0,1,1,1,1,0,1,0,1,1,1,1,1,0,0,1,1]
ghci> rebuildTree (compressTree (buildTree "suss"))
Node 0 (Leaf 'u' 0) (Leaf 's' 0)
```

Note that `rebuildTree . compressTree` is not the same as `id` because this compression is *lossy* and loses the original frequency counts. This function should, straightforwardly, perform a fixed amount of work per bit in the encoded tree ($O(n)$).

Finally, the following constants are defined in `src/Examples.hs`:

```
secretCode, secretTree :: [Int]
```

A special prize will be awarded to the first student who can a. demonstrate that their code passes ALL the given tests and b. decode the secret string with `decode (rebuildTree secretTree) secretCode`. Raise your hand if you think you've won!

Bonus: a better `reduce`

The `reduce` function from earlier makes use of `List.insert`, which makes the `buildTree` algorithm as a whole $O(n^2)$: sad. If `reduce` could be improved, we could get to the proposed complexity from earlier of $O(n \log n)$.

For a **"bonus" 2%**[2], import `Data.Set` and write a function `reduce' :: Ord a => Set (HTree a) -> HTree a` (and a corresponding `buildTree' :: Ord a => [a] -> HTree a` that uses it), which performs the same reduction algorithm but using a `Set` instead. You should look for relevant functions within `Data.Set`. The complexity of `reduce'` must be $O(n \log n)$ and it must be implemented tidily! **Do not modify the types of the existing pre-given functions!**

Start coding!

To begin working on your code, go to your **Home folder** where you will find a subfolder called **`compress`** which contains the **`src` skeleton folder**. A sample **`tests/Tests.hs`** file with some tests has also been provided. Good luck!

[2]The highest test score is obviously 100%, 102% is not possible, but getting the bonus will increase your mark a small amount if you didn't manage do everything else perfectly.