2020 Haskell January Test
Hash Array Mapped Tries

This test comprises three parts and the maximum mark is 30. There is one bonus mark available in Part I which will be added to your total, but marks will be capped at 30.

The **2020 Haskell Programming Prize** will be awarded for the best overall solution.

Credit will be awarded throughout for clarity, conciseness, *useful* commenting and the appropriate use of Haskell's various language features and predefined functions.

WARNING: The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.

1 Introduction

Most modern programming languages provide highly-tuned libraries for managing collections of objects. An example of a collection is a *set*, i.e. an assortment of unique objects in no particular order. Once you have sets, it's straightforward to build *maps* which are just sets of (key, value) pairs. One of the main challenges is to ensure that set operations, such as insertion, deletion and membership testing execute quickly, regardless of the size of the set. Another is to minimise the amount of memory used to store the set.

In this exercise you're going to implement a *hash array mapped trie*, or HAMT which is a simple, but ingenious, data structure for implementing sets that fulfils both objectives simultaneously. HAMTs are used by languages such as Scala, Clojure and Haskell to implement some of the more common set-related collection types.

2 Hash Array Mapped Tries (HAMTs)

A *trie* (from the word "re**trie**ve" and pronounced "tree") is a tree-like data structure for storing sets of values of some given type. To keep it simple we'll focus here exclusively on sets of integers, although the same principles can be applied to many other types.

Tries work by using the internal structure of each stored value to define a path through the trie from its root to a location where the value is stored. As we're working with integers the "internal structure" in this case will be the binary representation of those integers.

In general, tries can have an arbitrary number of levels, i.e. can have arbitrary depth, but in this exercise tries will have a specified maximum depth, which we will assume is at least 2. We'll number the levels from 0, so the root node is at level 0.

To illustrate this, Figure 1 shows a trie with maximum depth of 3 containing the integers 73, 206, 729, 1830 and 2521. Each internal *node*, shown as a circle, has sixteen (2^4) subtries, numbered 0...15, i.e. 0000...1111 in binary, and we arbitrarily number them from the *right*, as shown. The values in the set are stored in *leaves*, shown as boxes containing one or more values. Empty tries are shown as empty boxes, so you can think of these as leaves with no associated values.

In order to explain the structure of the trie we need to look at the binary representations of the values it contains, each of which is a 64-bit integer – the default for GHC on the lab machines. These are shown below using 12 bits to save space, as the remaining bits to the left are all 0. A gap has been added between each four-bit block to aid readability.

```
73:   0000 0100 1001
206:  0000 1100 1110
729:  0010 1101 1001
1830: 0111 0010 0110
2521: 1001 1101 1001
```

The least significant four bits of each value determines the index of the root subtrie containing the value. If this leads to a node (circle) then the next four most significant bits determine which of that node's subtries contains the value. The process continues until we reach a leaf, where the value is stored. Leaves can appear at any level other than level 0 (levels 1 and 2 in the example). However, only leaves at the lowest level can contain more that one value – the reason for this will become apparent when we look at how to build a trie (Section 4.3 below).

As an example, the values 206 and 1830 lead to subtries numbered 14 (1110_2) and 6 (0110_2) respectively, both of which are leaves containing the respective (single) values. The other three integers, i.e. 73, 729 and 2521, all have the same least significant four bits (1001) so they are all stored in the subtrie numbered $9 = 1001_2$. This is a reference to a node at the next level of the

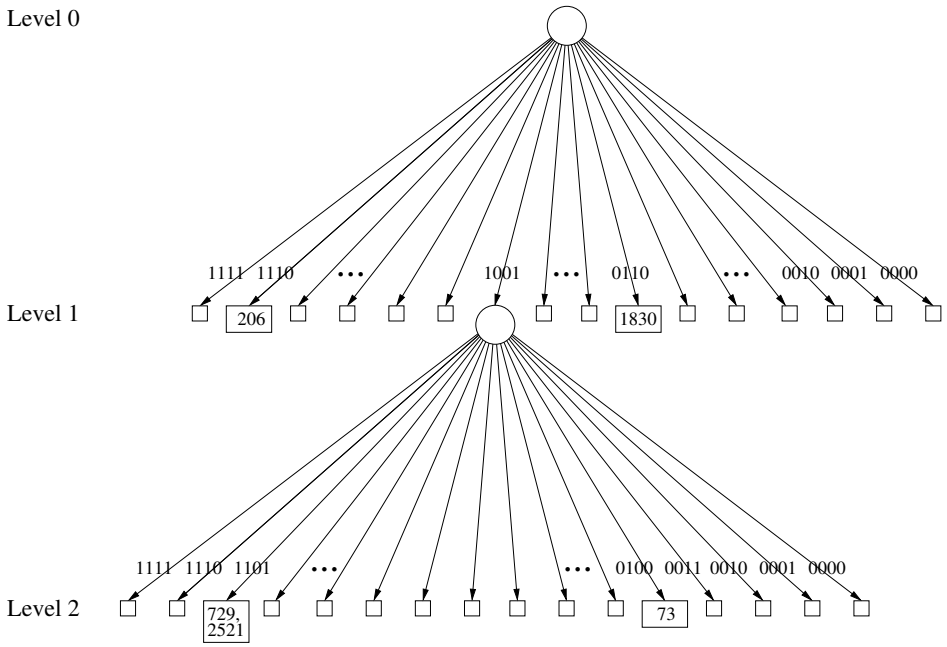

Figure 1: Example trie.

trie, so we therefore use the next four bits, bits 4-7, to index into that node's subtries. The value 73 leads to the subtrie numbered $4 = 0100_2$, which is a leaf containing just that value. The other two have the same bits in positions 4-7, i.e. 1101, and so end up in the same subtrie numbered $13 = 1101_2$. Because the trie has only three levels there can be no nodes further down the trie. We therefore say that the values 729 and 2521 "clash", in the sense that they cannot be distinguished using the least significant eight bits alone. Both values are therefore stored in the same leaf at the lowest level (level 2). Of course, we could add additional layers to the trie in the hope of avoiding clashes like this, but the example here is restricted to just three levels; this particular clash is therefore unavoidable.

2.1 Searching the trie

In order to determine whether a value, v say, is a member of a set represented by the above trie, we simply use the binary representation of v to define a path through the trie. If at any point we reach a leaf then v is a member of the set if and only if it is stored somewhere in the leaf. For example, if $v = 1830 = 01110010\underline{0110}_2$ then 0110 (underlined) directs the search to the subtrie (leaf) numbered $6 = 0110_2$ where the value 1830 is stored. The search and stored values match: success. If $v = 134 = 10000000\underline{0110}_2$, which has the same four least significant bits as 1830, we end up in the same subtrie as the previous search, but the value stored there is 1830≠134: failure. If $v = 1922 = 011110000\underline{010}$ the search ends at subtrie number $2 = 0010_2$ of the root, which is empty: failure. If $v = 2521 = 100\underline{111011001}$ then we need all eight least-significant bits to reach a leaf and this contains 729 and 2521: success.

2.2 Space efficiency

The problem with tries like Figure 1 is that most of the subtries are empty; collectively, they consume an enormous amount of memory that is essentially wasted. Rather than storing 16 subtries in each node regardless, it is much more space efficient to store only the non-empty ones, e.g. in an array: if there are three non-empty subtries, say, then the array will contain exactly three entries. But which of the 16 possible subtries do these three correspond to?

The trick is to add to each node a 16-bit *bit vector* and we can use a single (non-negative) integer to do this, because an integer can be thought of as a vector of bits (64 bits in the case of GHC). If the i^{th} bit of the integer is set, i.e. is equal to 1, then the i^{th} subnode is non-empty. So far so good, but which array element is associated with this bit? The answer comes from looking at the number of other '1' bits to its *right* in the bit vector; we could equally count bits to the left – the choice is arbitrary.

To illustrate this, Figure 2 (left) shows an equivalent trie to Figure 1 with this optimisation. Nodes are now shown as big circles. They contain a bit vector and an array of what we'll refer to as *subnodes*; each subnode is either a *terminal*, containing exactly *one* value, and shown as a single box, or a reference to a subtrie in the next level. If two or more values clash then they end up in a leaf at the lowest level of the trie, shown as a double box. In contrast to the trie structure in Figure 1, terminals (essentially singleton leaves) are now considered to be part of a node. By construction, leaves are now guaranteed to contain two or more values and collectively they make up the lowest level of the trie, as shown in the figure.

2.3 Modifying the search

Determining whether a value v is a member of the optimised trie of Figure 2 (left) now involves a little more work. First, we extract the least significant four bits, i.e. bits 0-3, of v. This will be a number, i say, between 0 and 15. Now use i to index the bit vector at the root of the trie. If this is 0 then v is not a member of the set and we're done. If it is 1 then the i^{th} subtrie is non-empty. To find its index in the subnode array, count the number of 1s, in the bit vector to the right of, but not including, index i and call this n, say. Now use n to index into the subnode array: if the subnode is a terminal then v is a member of the set if and only if the value at the terminal matches v; if the subnode is not a terminal then it refers to a subtrie in the next level, where the search continues.

The search scheme at the next level is identical to that at the root, except that we must use bits 4-7 of v. The last level of a trie – level 2 in the example here – comprises only leaves, each containing two or more values. If the search reaches a leaf then v is a member of the set if and only if it is contained in the list of values stored in the leaf.

As an example, the bit vector at the root of Figure 2 (left) is 0100001001000000. If $v = 206 = 11000000\underline{1110}$ then we inspect bit $14 = 1110_2$ (underlined) of the bit vector, which is 1. We therefore count the number of bits to the right of index 14 and there are two, so $n = 2$ in this case. We therefore index the subnode array at index 2. This references a terminal with value 206 and the search and stored values match: success. If, however, $v = 31 = 00000001\underline{1111}_2$ then the bit index is $15 = 1111_2$ and the corresponding bit in the vector is 0: failure. If $v = 73 = 01001001\underline{1001}$ then the bit index is 9 and the corresponding bit in the bit vector is 1. There is one bit to the right of this so $n = 1$ in this case. The subnode at index 1 is a reference to a node in the next level of the trie, where the search continues using bits 4-7 of v. The bit vector here is 0010000001000000. Bits 4-7 of v are $0100 = 4_{10}$ and the bit vector at index 4 is 1. There are no 1s to the right of this, so $n = 0$ in this case and we index the subnode array at 0. This references a terminal with value 73: success. If $v = 2521 = 100111011001$ then the search continues as for 73 until we get to level 1 of the trie. Now bits 4-7 are $1101 = 13_{10}$ and the corresponding bit in the bit vector is 1.

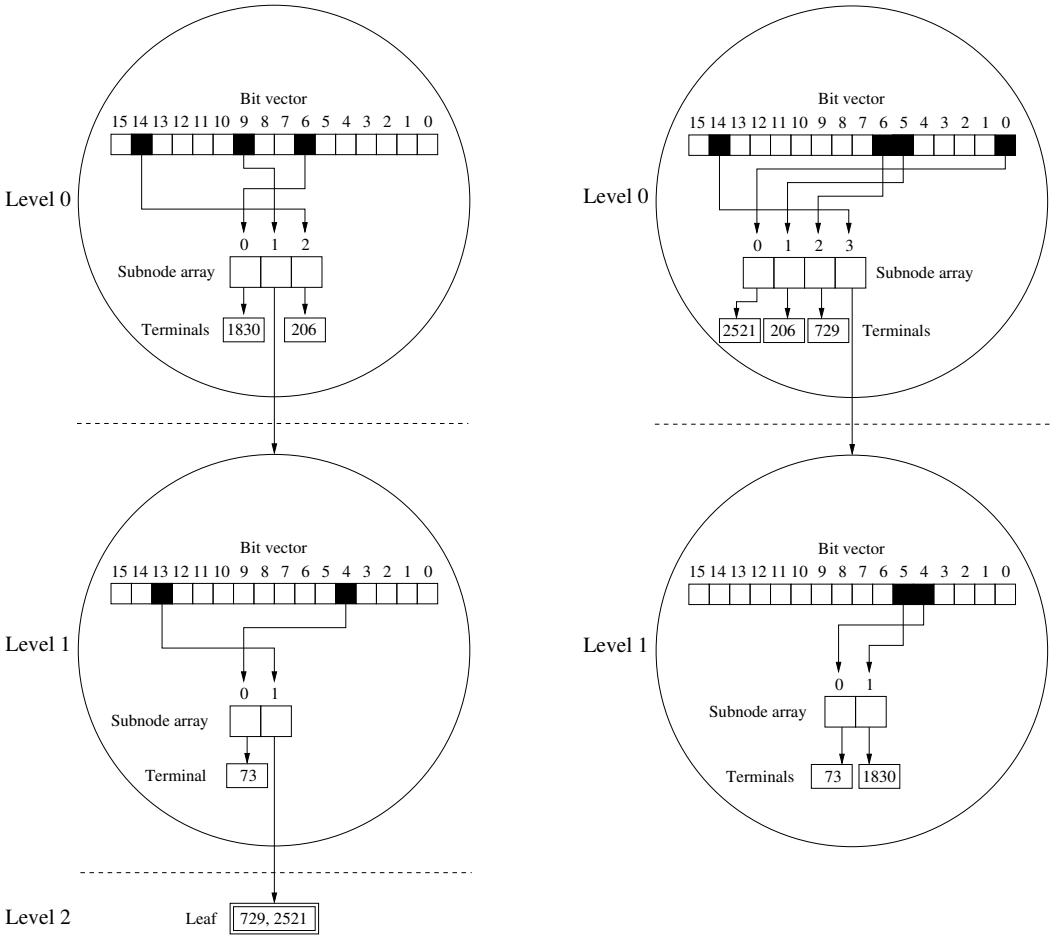

Figure 2: Array mapped tries indexed by raw (left) and hashed (right) values

There is a single 1 to the right of this, so we index the subnode array at index 1, which leads to a leaf containing the values 729 and 2521 in the lowest level of the trie. The search value $v = 2521$ matches one of these: success.

In general, if a trie contains more than three levels then the search process repeats similarly using successively more significant bits. Similarly, if there are more (or less) than a maximum of 16 subnodes at each level then more (or fewer) bits will be used at each level for indexing, as determined by the block size.

2.4 Hashing

One problem remains with the above scheme: in the worst case it is possible for the entire set to end up at the same leaf, whereupon the trie essentially degenerates into a list. A list is not a good structure for storing sets, as the search cost increases linearly with the size of the set. As an example, if the integers are are all multiples of 256 then their least significant eight bits will all be 0, so they will all clash in same leaf via subnode 0 in layer 1 and subnode 0 again in layer 2. Notice that choosing some other set of eight bits to do the indexing won't solve the problem.

For example, if we instead use the *most* significant bits, then all integers whose most significant bits are zero, e.g. the integers $0...N-1$ for some N, will end up in the same leaf. In short, we'd like to "spread the values out", ideally so that all values end up in terminals, rather than leaves; if they do end up in leaves then we'd like the number of clashes there to be small.

A very effective general solution to this problem is to use a *hash* of each integer, instead of the integer itself, to do the indexing. A *hashing function* maps an object to an integer and the idea is for these "hash values" to be spread out, so that objects that are grouped in some way in the space of values, e.g. numbers having all 0s in their least significant bits, become uniformly scattered in the corresponding space of hash values. The beauty of hash functions is that they can be applied to different object types[1], e.g. strings, lists, arrays etc., which makes it easy to generalise tries to store objects of any "hashable" type. To keep it simple, we will only focus on sets of integers in this exercise, so we only need to consider hash functions that map integers to integers.

Figure 2 (right) shows the trie produced when using a hashing function that maps the values 2521,73, 206, 729 and 1830 to 391636870159797326, 1105127178103142997, 3910943446213641574, 8629109329697440862 and 4922377257571046896 respectively. Notice that the values are now all stored at terminals inside nodes, i.e. there are no clashes and hence no leaves. Bingo!

3 Representing HAMTs

A fully optimised implementation of HAMTs in Haskell would use Haskell arrays to store the subnodes at each node. However, because arrays are not covered in the course we'll instead mock them up using lists. A Java-like array access like `a[n]` will be implemented here via `a !! n`, where `a` is a list. Don't won't worry about the fact that `!!` isn't a constant-time operation, as it's straightforward to replace lists with arrays, should we wish to make the implementation more efficient.

With this in mind HAMTs (Haskell type `Trie`) can be implemented using the following types, which should be self-explanatory:

```
type BitVector = Int

data Trie = Leaf [Int] | Node BitVector [SubNode]
          deriving (Eq, Show)

data SubNode = Term Int | SubTrie Trie
             deriving (Eq, Show)

type Hash = Int

type HashFun = Int -> Hash
```

Note that each `SubNode` is either a (`Term`)inal value, i.e. a single `Int`, or a `SubTrie`, i.e. a reference to a `Trie` in the next level[2].

Note also that integers in GHC are 64-bits long, so if we use them to implement bit vectors, as here, then each node in a trie can have a maximum of $64 = 2^6$ subnodes, i.e. the maximum

[1] There are infinitely many possible hashing functions for a given object type and picking a "good" one is an art in itself.

[2] Note that a smart implementation in C or C++, for example, might use a single bit to distinguish terminal values from `SubTrie` references, enabling values to be stored directly in the array, but we won't worry about that in this exercise.

block size will be $b = 6$ bits. In order to simplify the manipulation of the bits of an integer the `Data.Bits` module has been imported in the skeleton file; a crib sheet is provided separately.

4 What to do

There are three parts to this exercise. Part III is tricky, as you have to work out some of the details yourself, so you are advised to complete Parts I and II before attempting Part III.

To help with testing, a number of example tries are included in the `Examples` module, including those corresponding to Figure 2, i.e. indexed with both raw and hashed values respectively:

```
figure :: Trie
figure
  = Node 16960 [Term 1830,
                SubTrie (Node 8208 [Term 73,
                                    SubTrie (Leaf [729,2521])]),
                Term 206]

figureHashed :: Trie
figureHashed
  = Node 16481 [Term 2521,
                Term 206,
                Term 729,
                SubTrie (Node 48 [Term 73,
                                  Term 1830])]
```

A show function, `showT` is also included in the `Types` module for displaying tries. Nodes are displayed starting with the bit vector, which is shown as a 16-bit string. The subnodes are then shown indented in index order, starting from 0. Terminal values within nodes (of the form `Term v`) are shown within <angle brackets> and leaves are displayed as Haskell lists, i.e. using [square brackets]. Deeper levels of nesting are reflected in the indentation. For example:

```
*Tries> showT figure
0100001001000000
  <1830>
  0010000000010000
    <73>
      [2521,729]
  <206>
*Tries> showT figureHashed
0100000001100001
  <2521>
  <206>
  <729>
  0000000000110000
    <73>
    <1830>
```

Note that `showT` assumes that subnodes are indexed using a maximum of four bits, so each bit vector is displayed using 2^4=16 bits. However, your code should accommodate any valid block size.

A function `showBitVector` is also provided that computes the binary representation of an integer using a specified number of bits. The result is a string, e.g.

```
*Tries> showBitVector 15 5
"01111"
*Tries> showBitVector 1024 11
"10000000000"
```

4.1 Part I: Utilities

1. Define a function `countOnes :: Int -> Int` that will count the number of 1 bits in the binary representation of a given integer. You can process the bits one at a time using `div` and `mod` for full marks. Alternatively, you can process them in batches of four bits at time using the following table, defined in the template, which should be faster:

   ```
   bitTable :: [Int]
   bitTable
     = [0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4]
   ```

   Here, element $0 \leq i \leq 15$ is the number of bits in the integer i. There is a bonus of 1 mark for the batched solution. For example,

   ```
   *Tries> countOnes 0
   0
   *Tries> countOnes 65219
   11
   ```

   Note that there are much faster ways to solve the problem, e.g. Kernighan's algorithm which is used in the equivalent `popCount` function in `Data.Bits`[3]. However, please do not write `countOnes = popCount`, or similar!

   [2 (+1) Marks]

2. Using `countOnes` above, or `popCount` from `Data.Bits`, define a function `countOnesFrom :: Int -> Int -> Int` such that `countOnesFrom i n` returns the number of 1 bits to the *right* of, i.e. not including, bit i in the binary representation of n. The bits are numbered from the right, so index 0 corresponds to the least significant bit. For example,

   ```
   *Tries> countOnesFrom 0 88
   0
   *Tries> countOnesFrom 3 15
   3
   ```

   Hint: Use `.&.` and `bit` from `Data.Bits` to mask the least significant i bits of `n`.

   [2 Marks]

3. Define a function `getIndex :: Int -> Int -> Int -> Int` that will extract a specified number of bits, given by the block size (third argument), from a given integer (first argument), returning the result as an integer. For example, if the given integer is $53767 = 1101\ 0010\ 0000\ 0111_2$ then there are four blocks of four bits in its 16-bit representation, as

[3] You might like to time your solution against `popCount`: you will lose!

highlighted by the spacing, with the rightmost being block 0. The second argument specifies which of those blocks to choose. In practice, when manipulating tries, this argument will represent the current trie level. Thus, for example:

```
*Tries> showBitVector 53767 16
"1101001000000111"
*Tries> getIndex 53767 0 4
7
*Tries> getIndex 53767 3 4
13
*Tries> getIndex 53767 1 8
210
```

(the topmost 8 bits of 53767 are $11010010 = 210_{10}$).

[2 Marks]

4. Define a function `replace :: Int -> [a] -> a -> [a]` that will replace the n^{th} item (first argument) of a given list with a given value. For example,

```
*Tries> replace 0 "trie" 'b'
"brie"
*Tries> replace 3 "trie" 'p'
"trip"
```

A precondition is that the index is less than the length of the list.

[3 Marks]

5. Define a function `insertAt :: Int -> a -> [a] -> [a]` that will insert a given item at a given position (index) in a given list. For example:

```
*Tries> insertAt 3 'p' "trie"
"tripe"
*Tries> insertAt 4 's' "trie"
"tries"
```

A precondition is that the index is less than or equal to the length of the list.

[3 Marks]

4.2 Part II: Functions on HAMTs

1. Define a function `sumTrie :: (Int -> Int) -> ([Int] -> Int) -> Trie -> Int` that will 'sum" the components of a trie. By "summing" the components we might mean computing the number of values stored in the trie, the sum of those values, or the total number of "bins", i.e. the sum of the number of `Term`inals and `Leaf`s. The two function-valued arguments allow these, and possibly other, notions of "sum" to be specified. The rules are:

   - To sum a `Term`inal, apply the first function to the associated value (an `Int`).
   - To sum a `Leaf`, apply the second function to the list of elements stored at the leaf (a `[Int]`).
   - To sum a `Node`, recursively sum the elements in its list of `SubNode`s.
   - To sum a `SubTrie`, apply `sumTrie` to the `Trie` it refers to.

Note that supplying `(const 1)` and `length` as the two functions causes `sumTrie` to compute the number of values stored in the given trie and supplying `(const 1)` and `(const 1)` causes it to compute the number of bins. You will find three functions commented out in the template that make use of `sumTrie`:

```
trieSize :: Trie -> Int
binCount :: Trie -> Int
meanBinSize :: Trie -> Double
```

The function `meanBinSize` computes the average number of values in each "bin" – this is one way of measuring the effectiveness of the hashing function in spreading values throughout the trie. If you wish you can uncomment these when you've defined `sumTrie`, as they may be useful in testing. Thus, for example:

```
*Tries> sumTrie (const 1) length empty
0
*Tries> trieSize empty
0
*Tries> sumTrie (const 1) (const 1) figure
4
*Tries> binCount figure
4
*Tries> meanBinSize figure
1.25
*Tries> meanBinSize figureHashed
1.0
```

The constant `empty = Node 0 []` is defined in the template[4].

[4 Marks]

2. Using `getIndex` and `countOnesFrom` define a function `member :: Int -> Hash -> Trie -> Int -> Bool` that will determine whether a given value (first argument) is a member of a given trie. In order to do this you need to use the hash of the value (second argument) to traverse the trie. The fourth argument is the block size. For testing purposes the imported module `HashFunctions` contains a function `hash :: HashFun`, equivalent to `hash :: Int -> Hash`, that generates a hash value from a given integer, e.g.

```
*Tries> hash 1
5364745905576871
*Tries> hash 73
391636870159797326
```

You don't need to understand how the function works. For example (recall that `figure` is built using the raw values to index the trie, rather than their hashed values):

```
*Tries> member 12 12 figure 4
False
*Tries> member 73 73 figure 4
```

[4] Recall that the smallest maximum depth is 2, so all tries have at least one node at their root.

```
True
*Tries> member 206 (hash 206) figureHashed 4
True
```

Hints: Use a helper function, and carry round the current level (initially 0) as you traverse the trie. You'll need this when invoking `getIndex` to make sure you pick out the correct bits from the hashed value (`Hash`) to index into the subnode array. You can use `testBit` from `Data.Bits` to test a specified bit in the bit vector associated with a node.

[6 Marks]

4.3 Part III: Tree building

To complete the exercise you're now going to build a trie from a given list of integers.

1. Define a function `insert :: HashFun -> Int -> Int -> Int -> Trie -> Trie` that will deliver the result of inserting a value (fourth argument) into a trie. The first argument is the hashing function – you may need this to re-compute the hash of a value that's already in the trie (see below). The second and third arguments are the maximum trie depth and block size respectively.

   You can build the trie any way you see fit, but are strongly advised to follow the scheme below for inserting a value, `v` say:

   - You need to keep track of which level you're inserting into, so carry this round. Recall that the top level is indexed 0.
   - If you reach a `Leaf` then add `v` to the elements in the leaf, unless `v` is already there, in which case return the leaf unmodified. Note that the order of the elements within a `Leaf` list is unimportant.
   - If you reach the lowest level of the trie (maximum depth - 1) then return a `Leaf` containing the singleton list `[v]`. Wait: aren't leaves supposed to contain at least *two* values? Don't worry – they will (see below)!
   - If you encounter a `Node` then inspect the bit vector at the required index (see Section 2.3). If this is 0 then build a new node with the bit set to 1 and use `insertAt` to add a new `Term` containing `v` to the subnode array. If it is already 1 then use `replace` to replace the corresponding element of the subnode array with a new `SubNode`. The rules are:
     - If the current `SubNode` is a `SubTrie` then insert `v` into the subtrie.
     - If the current `SubNode` is of the form `Term v'` then then return it unmodified if `v == v'`; otherwise insert `v'` into an empty trie and then insert v into the result you get back. This is where you need the `HashFun`, as you need to re-compute the hash for v' in order to insert it. Notice that we're doing *two* insertions here so if we've hit the lowest level of the trie we'll end up with a leaf containing *two* elements when we're done.

   You clearly need a helper function to carry stuff around, but what should its type be?

   For example:

```
*Tries> let t1 = insert id 3 4 2521 empty
*Tries> showT t1
```

```
0000001000000000
  <2521>
*Tries> let t2 = insert id 3 4 1830 t1
*Tries> showT t2
0000001001000000
  <1830>
  <2521>
*Tries> let t3 = insert id 3 4 729 t2
*Tries> showT t3
0000001001000000
  <1830>
  0010000000000000
      [729,2521]  <--- Note: any order will do here
*Tries> let t4 = insert hash 3 4 729 empty
*Tries> let t5 = insert hash 3 4 2521 t4
*Tries> let t6 = insert hash 3 4 206 t5
*Tries> showT t6
0000000001100001
  <2521>
  <206>
  <729>
```

Note that using `id` as the hashing function has the effect of using the raw integers themselves to perform trie indexing.

[7 Marks]

2. Use a `fold` to define a function `buildTrie :: HashFun -> Int -> Int -> [Int] -> Trie` that uses `insert` to build a trie from a given list of integer values. The `HashFun` and two `Int` arguments are the same as the first three arguments of `insert`. Again, the order of the elements with a `Leaf` list is unimportant. For example:

```
*Tries> let t1 = buildTrie id 3 4 [1..3]
*Tries> showT t1
0000000000001110
  <1>
  <2>
  <3>
*Tries> let t2 = buildTrie id 3 4 [n * 256 | n <- [1..5]]
*Tries> showT t2
0000000000000001
  0000000000000001
      [256,512,768,1024,1280]
*Tries> buildTrie id 3 4 [73,206,729,1830,2521] == figure
True
*Tries> buildTrie hash 3 4 [73,206,729,1830,2521] == figureHashed
True
```

[1 Mark]

2020 Haskell January Test
Supplementary notes

1 Bit twiddling

This year's Haskell test will involve some bit manipulation, which is sometimes referred to as "bit twiddling". To make this easy you'll be invited to use some of the functions from the Haskell module `Data.Bits`. The objective of these notes is to give you a brief introduction to the functions in `Data.Bits` and explain how to use them to implement some common bit twiddling operations.

Module `Data.Bits`

The `Data.Bits` module contains a type class which starts off like this:

```
class Eq a => Bits a where ...
```

For each type that is an instance of `Bits` the idea is to be able to manipulate objects of that type as *bit vectors*. The only instance of interest to the test is that for `Int`. The version of GHC you will be using supports 64-bit integers, so every object of type `Int` can be thought of as being a vector of 64 bits. For example, the integer 29855 has the following bit vector (binary) representation:

```
0000000000000000000000000000000000000000000000000111010010011111
```

Wow - that's a lot of bits! To keep things a little easier to write out in these notes we'll use examples containing "small" integers so we only have to focus on the least significant n bits for some "small" n. For example, we can display 29855 using as few as 15 bits, as it's most significant bit is at index 14 (note that bit indices start at 0). Here it is written out with 16 bits:

```
0111010010011111
```

The remaining 48 bits are there, of course, and can be used if necessary, so you shouldn't forget them.

To help you to practice with `Data.Bits` a file `practice.hs` has been uploaded to the Materials page. This imports `Data.Bits` and implements a useful function, `showBitVector :: Int -> Int -> String`, for displaying bit vector representations of integers. The first argument is the integer (bit vector) to display and the second specifies the number of bits to generate in the resulting string. For example:

```
*Main> showBitVector 29855 16
"0111010010011111"
```

The member functions of `Data.Bits` include the following, which we'll explain with examples. Each of these is implemented directly in low-level machine code instructions, so are super-fast. Somewhat unusually, we'll be quite interested in performance in this year's test.

Bit-wise "and" and "or"

The operators `(.&.) :: a -> a -> a` and `(.|.) :: a -> a -> a` respectively implement bit-wise "and" and "or" on the bit vector representation of objects of type `a`. Here, we're only interested in the `Int` instance where the operators have the types:

```
(.&.) :: Int -> Int -> Int
(.|.) :: Int -> Int -> Int
```

We'll apply a similar instantiation when describing the other member functions below.

The expression `i .&. j` performs a boolean (2-way) "and" on each bit of `i` and `j` pair-wise. The "and" works similarly to Haskell's built-in `&&` operator, except that it works on bits (0 and 1) rather than booleans (`False` and `True`). Similarly, `i .. j`— applies "or" pair-wise, where "or" works similarly to `||`. For example:

```
*Main> showBitVector 9 4
"1001"
*Main> showBitVector 5 4
"0101"
*Main> 9 .&. 5
1
*Main> showBitVector (9 .&. 5) 4
"0001"
*Main> 9 .|. 5
13
*Main> showBitVector (9 .|. 5) 4
"1101"
```

`shiftL` and `shiftR`

The functions `shiftL :: Int -> Int -> Int` and `shiftR :: Int -> Int -> Int` shift all the bits of a bit vector left/ right respectively. The first argument is the bit vector (integer) and the second is the number of positions to shift. Zeros are added at the right/left bit positions accordingly. For example:

```
*Main> showBitVector 2000 16
"0000011111010000"
*Main> showBitVector (shiftL 2000 2) 16
"0001111101000000"
*Main> shiftL 2000 2
8000
*Main> showBitVector (shiftR 2000 3) 16
"0000000011111010"
*Main> shiftR 2000 3
250
```

Remark: Although it's not important, you might note that shifting left/right by n positions is equivalent to multiplying/dividing by 2^n using integer arithmetic.

`bit`

The function `bit :: Int -> Int` generates an integer whose bit vector representation has a 1 at the specified bit index (`Int`) and 0s elsewhere. The least significant bit has index 0. For example:

```
*Main> bit 0
1
*Main> showBitVector (bit 0) 8
"00000001"
*Main> bit 5
32
*Main> showBitVector (bit 5) 8
"00100000"
```

Note that `bit n` is equivalent to $2^{\texttt{n}}$.

`setBit` and `clearBit`

The functions `setBit :: Int -> Int -> Int` and `clearBit :: Int -> Int -> Int` respectively set and clear the bit at the specified index (second argument) in a given bit vector (first argument). For example:

```
*Main> showBitVector 39855 16
"1001101110101111"
*Main> setBit 39855 14
56239
*Main> showBitVector (setBit 39855 14) 16
"1101101110101111"
*Main> clearBit 39855 0
39854
*Main> showBitVector (clearBit 39855 0) 16
"1001101110101110"
```

`testBit`

The function `testBit :: Int -> Int -> Bool` returns `True` if the bit at the specified index (second argument) in a given bit vector (first argument) is 1; `False` otherwise. For example:

```
*Main> showBitVector 1771 12
"011011101011"
*Main> testBit 1771 3
True
*Main> testBit 1771 8
False
```

1.1 Masking

A common operation on bit vectors is *masking*, which involves using `.&.`, often in conjunction with `bit` and `shiftL/R`, to zero out all but a specified set of bits in a bit vector. First, it's useful to note that `bit n - 1` $= 2^{\texttt{n}} - 1$ is a bit vector with 1s in positions 0...`n-1` and 0s elsewhere; it's often referred to as a *mask* because "anding" it with a given bit vector has the effect of zeroing, i.e. masking out, all bits of the bit vector where the mask is 0. For example:

```
*Main> showBitVector (bit 6 - 1) 16
"0000000000111111"
*Main> showBitVector 44628 16
```

```
"1010111001010100"
*Main> showBitVector (44628 .&. (bit 6 - 1)) 16
"0000000000010100"
```

This gives you the least significant 6 bits of 44628. Any other contiguous set of bits can be extracted by applying `shiftR`, for example, the expression `shiftR 44628 5 .&. (bit 6 - 1)` extracts bits 5-10 (underlined) of 44628=$1010\underline{110010}10100_2$, returning the result $50 = 110010_2$ as an integer:

```
*Main> showBitVector 44628 16
"1010111001010100"
*Main> showBitVector (shiftR 44628 5) 16
"0000010101110010"
*Main> showBitVector (bit 6 - 1) 16
"0000000000111111"
*Main> shiftR 44628 5 .&. (bit 6 - 1)
50
*Main> showBitVector (shiftR 44628 5 .&. (bit 6 - 1)) 16
"0000000000110010"
```

That's all you need for now. Get practising! These notes will be made available to you during the test.