The University of Hong Kong

COMP3258: Functional Programming

Assignment 2

Deadline: 23:59, Nov 14, 2022 (HKT)

---

1. Please do not import other modules in your code.
2. We encourage to use the template file provided. And please do not modify the type signatures we provided.
3. For absurd cases (don't be confused with base case), feel free to raise exception or give a default value for it.
4. Please submit a single Haskell file, named as `A2_XXX.hs`, with `XXX` replaced by your UID, and follow all the type signatures strictly. In the case that Moodle rejects your .hs file, please submit it as `A2_XXX.zip`, which includes only one file `A2_XXX.hs`.
5. Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment, **and explain it in a detailed manner**. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.

0 Warm up (15 pts)

The first problem involves writing some functions for a tree data structure, the one that we have discussed during classes and tutorials. The following datatype defines a binary tree, storing data at each internal node.

```haskell
data Tree a = Leaf
            | Branch a (Tree a) (Tree a)
    deriving (Show, Eq)
```

Now consider definitions of `fold` and `map` for this data structure.

```
foldTree :: b -> (a -> b -> b -> b) -> Tree a -> b
foldTree e _ Leaf = e
foldTree e n (Branch a n1 n2) = n a (foldTree e n n1) (foldTree e n n2)

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f = foldTree Leaf (\x t1 t2 -> Branch (f x) t1 t2)
```

Use these two functions to:

Problem 1. (5 pts) Implement `takeWhileTree`, applied to a predicate `p` and a tree `t`, returns the largest prefix tree of `t` (possibly empty) where all elements satisfy `p`.

```
*Main> tree1 = Branch 1 (Branch 2 Leaf Leaf) (Branch 3 Leaf Leaf)
*Main> takeWhileTree (< 3) tree1
Branch 1 (Branch 2 Leaf Leaf) Leaf
*Main> takeWhileTree (< 0) tree1
Leaf
*Main> tree2 = Branch 4 (Branch 2 Leaf Leaf) (Branch 3 Leaf Leaf)
*Main> takeWhileTree (< 3) tree2
Leaf
```

Problem 2. (10 pts) Implement `zipTree f xs ys` that returns the tree obtained by applying `f` to each pair of corresponding elements of `xs` and `ys`. If one branch is longer than the other, then the extra elements are ignored.

```
*Main> zipTree (+) (Branch 1 Leaf (Branch 2 Leaf Leaf)) (Branch 3 Leaf Leaf)
(Branch 4 Leaf Leaf)
```

1 Propositional Logic (25 pts)

In this problem, we're going to implement parts of propositional logic, where propositions are joined with logical connectives. In our setting, the syntax is defined as follows:

```haskell
type Name = String
data Prop = Var Name
          | F
          | T
          | Not Prop
          | Prop :|: Prop
          | Prop :&: Prop
          | Prop :->: Prop
          | Prop :<->: Prop
```

In this datatype definition:

- The propositions are represented as `Prop` type.
- We use `Var` to create propositional variables (e.g., `Var "P"`);
- use `F` to build a logic false value (don't be confused with Haskell `False` value);
- use `T` to build a logic true value;
- use `Not` to build a negation (e.g., `Not (Var "P")`);
- use `:|:` and `:&:` to construct disjunction and conjunction.
- `:->:` is implication and `:<->:` is bi-implication ("if and only if").

We provide the truth table of the last two constructors to avoid misunderstanding.

P	Q	P -> Q	P <-> Q
F	F	T	T
F	T	T	F
T	F	F	F
T	T	T	T

Problem 3. (10 pts) We say that proposition A is

- Satisfiable, if there is some truth assignment for the variables of A that satisfies A (make A be true);
- Unsatisfiable, if no truth assignment satisfies A
- Valid (or tautology), if every truth assignment satisfies A

Your task is to define three functions to detect whether a proposition is satisfiable, unsatisfiable or valid.

```haskell
satisfiable :: Prop -> Bool
satisfiable = undefined

unsatisfiable :: Prop -> Bool
unsatisfiable = undefined

valid :: Prop -> Bool
valid = undefined
```

Expected Results:

```
*Main> satisfiable $ Var "P"
True
*Main> unsatisfiable $ (Var "P") :&: (Not (Var "P"))
True
*Main> valid $ (Var "P") :|: (Not (Var "P"))
True
```

Problem 4. (5 pts) Define an `eval` function, given an environment (contains a list of names associated with their truth values) and a proposition, and return a boolean to indicate its truth.

```haskell
type Env = [(Name, Bool)]

eval :: Env -> Prop -> Bool
eval = undefined
```

Problem 5. (10 pts) Disjunctive normal form (DNF) is a canonical normal form of a logical formula consisting of a disjunction of clauses, where a clause is a conjunction of literals (variables and negated variables).

For example, all of the following formulas are in DNF:

```haskell
-- (A /\ ~B /\ ~C) \/ (~D /\ E /\ F)
dnf1 = ((Var "A") :&: (Not (Var "B")) :&: (Not (Var "C")))
           :|: ((Not (Var "D")) :&: (Var "E") :&: (Var "F"))

-- (A /\ B) \/ (C)
dnf2 = ((Var "A") :&: (Var "B")) :|: (Var "C")
```

```haskell
-- (A /\ B)
dnf3 = (Var "A") :&: (Var "B")
```

In Haskell, we can take a slightly different way to store DNF, we use lists of lists. For the following propositions, we can have

```haskell
-- (A /\ ~B /\ ~C) \/ (~D /\ E /\ F)
dnf1 = [[Var "A", Not (Var "B"), Not (Var "C")],
        [Not (Var "D"), Var "E", Var "F"]]

-- (A /\ B) \/ (C)
dnf2 = [[Var "A", Var "B"],
        [Var "C"]]

-- (A /\ B)
dnf3 = [[Var "A", Var "B"]]
```

Your task is to implement a function `toDNF` that takes a `Prop` and return a `[[Prop]]` which is in DNF.

```haskell
toDNF :: Prop -> [[Prop]]
toDNF = undefined
```

You can use this link to check the examples of DNF.

2 Trie (30 pts)

Trie, is a type of search tree, a tree data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters. To access a key (to recover its value, change it, or remove it), the trie is traversed depth-first, following the links between nodes, which represent each character in the key.

Unlike a binary search tree, nodes in the trie do not store their associated key. Instead, a node's position in the `Trie` defines the key with which it is associated. This distributes the value of each key across the data structure and means that not every node necessarily has an associated value. All the

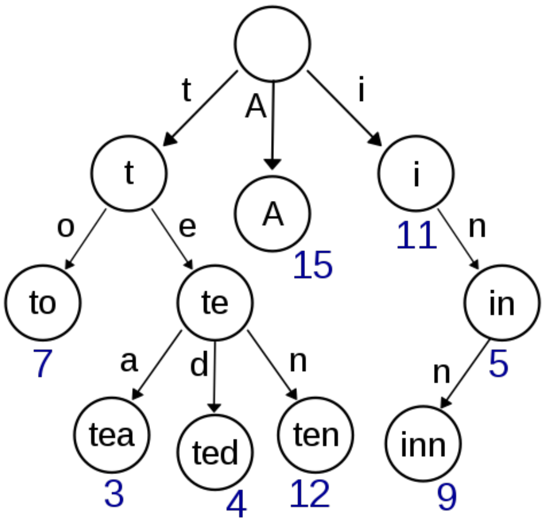


Figure 1: An example of Trie, whose key is `String` and value is `Int`

children of a node have a common prefix of the string associated with that parent node, and the root is associated with the empty string.

Though tries can be keyed by character strings, they need not be. The same algorithms can be adapted for ordered lists of any underlying type, e.g. permutations of digits or shapes. In particular, a bitwise trie is keyed on the individual bits making up a piece of fixed-length binary data, such as an integer or memory address. An example of `Trie` is shown below:

We have provided the type definition of `Trie` as follows. Each trie node contains a `value` and a list of (`element of key`, `trie`) tuples.

```haskell
data Trie k v = Trie { value :: Maybe v
                     , subTrees :: [(k, Trie k v)]} deriving (Show)
```

To clarify, `key` is of type `[k]` and `element of key` is of type `k`.

We also provide the method to generate an empty trie, which does not contain any value (by using `Nothing` as its value) and sub-tries.

```haskell
emptyTrie :: Trie k v
emptyTrie = Trie Nothing []
```

Your tasks are implementing several functions, including `insertTrie`, `lookupTrie`, `fromList` and `fromString`.

Problem 6. (10 pts) Implement a function `insertTrie` to insert a value.

```
*Main> insertTrie emptyTrie [0] 0
Trie {value = Nothing, subTrees = [(0,Trie {value = Just 0, subTrees = []})]}
*Main> insertTrie (insertTrie emptyTrie [0, 1] 0) [0, 2] 1
Trie {value = Nothing, subTrees = [(0,Trie {value = Nothing,
  subTrees = [(1,Trie {value = Just 0, subTrees = []}),
              (2,Trie {value = Just 1, subTrees = []})]})]}
```

Problem 7. (5 pts) Implement a function `lookupTrie` that finds the value from a `Trie` given a key. If the key does not exist in the `Trie`, return `Nothing`.

Problem 8. (5 pts) Implement a function `fromList :: Eq k => [([k], v)] -> Trie k v` that builds a `Trie` from a list of `(key, value)`.

Notice: If there are two identical keys in the `List`, the value of the latter one will override the former one.

```
*Main> fromList []
Trie {value = Nothing, subTrees = []}
*Main> fromList [([1],1), ([1],2)]
Trie {value = Nothing, subTrees = [(1,Trie {value = Just 2, subTrees = []})]}
```

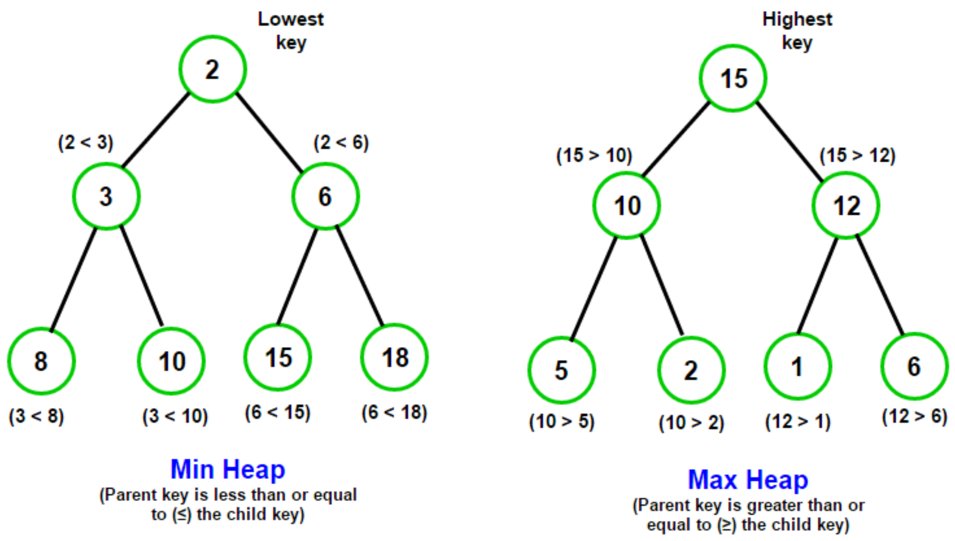
Problem 9. (10 pts) Implement a function `fromString` that builds a `Trie` from a `String` of a sentence, using words separated by white space characters as key sequences and their indices (in the sentence) as value.

```
*Main> fromString "a ab"
Trie {value = Nothing,
      subTrees = [('a',Trie {value = Just 1,
                             subTrees = [('b',Trie {value = Just 2,
                                                    subTrees = []})]})]}
*Main> fromString "a\nab"
Trie {value = Nothing,
      subTrees = [('a',Trie {value = Just 1,
                             subTrees = [('b',Trie {value = Just 2,
                                                    subTrees = []})]})]}
```

3 Functional Data Structure (30 pts)

In this section, we're going to learn by stepping through constructing data structure functionally and utilising this data structure to solve challenging problems in a practical sense.

Problem 10. (10 pts) Heap is a specialized tree-based data structure that satisfies the heap property: In a *min-heap*, if P is a parent node of C, the key of P is less than or equal to the key of C. The node at the "top" of the heap is called the root node. We show a sample of a min-heap in the picture.



Heapsort is a sorting algorithm that uses the heap data structure to sort a list. We can easily obtain a sorted list by flattening a heap.

We reuse the `Tree` definition in the first section. The function `flatten` converts a heap (a tree satisfying the heap property) into a list; the function `merge` combines two sorted list into a sorted list.

```
data Tree a = Leaf
            | Branch a (Tree a) (Tree a)
    deriving (Show, Eq)

flatten :: Ord a => Tree a -> [a]
flatten Leaf = []
flatten (Branch x l r) = x : merge (flatten l) (flatten r)
```

```haskell
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) | x <= y = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys
```

Our task is to implement the function `buildHeap` that builds a min-heap from a raw (unsorted) list, and then compose those defined functions to implement a sorting algorithm `heapSort`.

```haskell
heapSort :: Ord a => [a] -> [a]
heapSort = flatten . buildHeap
```

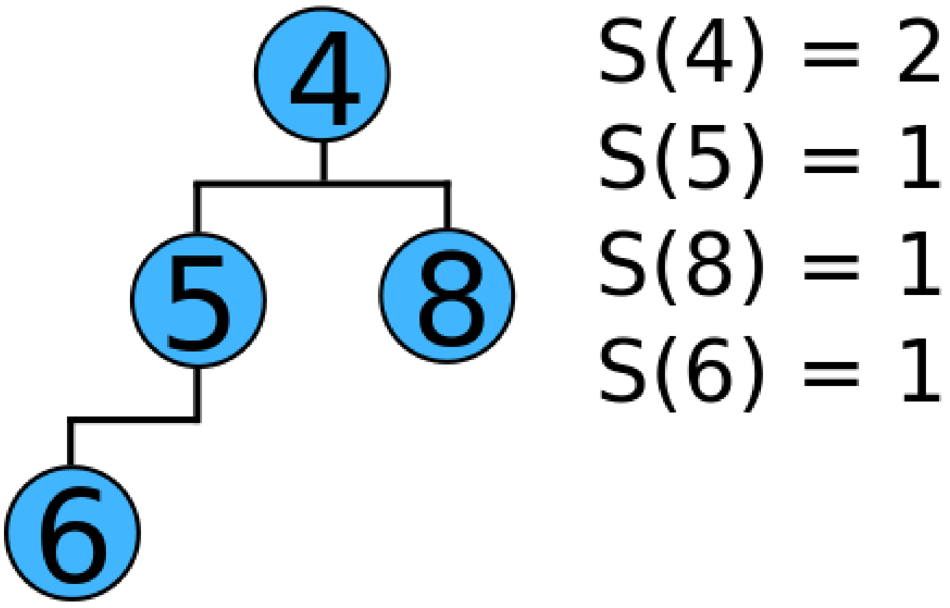
The function `buildHeap` can be split into two parts: `buildTree` is to build a raw tree from a given list; `heapify` is to convert a tree into another tree that satisfies the heap property. The `heapify` function is based on `siftDown` function that moves the value *down* the tree by successively exchanging the value with the bigger of its two children. The operation continues until the value reaches a position where it is less than both its children, or, failing that until it reaches a leaf.

```haskell
buildHeap :: Ord a => [a] -> Tree a
buildHeap = heapify . buildTree

buildTree :: Ord a => [a] -> Tree a
buildTree = undefined

heapify :: Ord a => Tree a -> Tree a
heapify Leaf = Leaf
heapify (Branch x l r) = siftDown x (heapify l) (heapify r)

siftDown :: Ord a => a -> Tree a -> Tree a -> Tree a
siftDown = undefined
```

Please implement two functions `buildTree` and `siftDown`. And justify the implementation by yourself using `heapSort`.

Problem 11. (20 pts) A priority queue is an abstract data type similar to a regular queue data structure in which each element additionally has a priority associated with it. The critical property of a priority queue is that the priority of an element at a node is no larger than the priorities in each of its subtrees (similar to our min-heap, but with priorities). We can give an implementation of a priority queue using a *leftist heap*, which makes the merge operations efficient.

Leftist heap has the extra property that “the order of the left subtree of any node is no smaller than the rank of its right subtree”. We define `Order` as the length of the shortest path in the tree from the root to a *Null* tree. We show how to compute the order value (aka S-value) below.



```
data PQueue a p = Null
                | Fork Order a p (PQueue a p) (PQueue a p) deriving (Show, Eq)
type Order = Int
```

The essential interface of the priority queue is two functions: `insert` and `delete` which takes at most logarithmic time in the length of the list.

```
insert :: Ord p => a -> p -> PQueue a p -> PQueue a p
delete :: Ord p => PQueue a p -> ((a, p), PQueue a p)
```

First, similar to our previous question, we can flatten the queue into a list and the result is a sorted list according to the priority.

```haskell
flattenQ :: Ord p => PQueue a p -> [(a, p)]
flattenQ = undefined
```

Expected Results

```haskell
sampleTree1 :: PQueue String Int
sampleTree1 = Fork 2 "A" 4 (Fork 1 "B" 5 (Fork 1 "C" 6 Null Null) Null)
                           (Fork 1 "D" 8 Null Null)

sampleTree2 :: PQueue String Int
sampleTree2 = Fork 2 "A" 1 (Fork 2 "B" 3 (Fork 1 "C" 4 Null Null)
                                         (Fork 1 "D" 6 Null Null))
                           (Fork 1 "E" 10 (Fork 1 "F" 11 Null Null) Null)
```

```
Main> flattenQ sampleTree1
[("A",4),("B",5),("C",6),("D",8)]
Main> flattenQ sampleTree2
[("A",1),("B",3),("C",4),("D",6),("E",10),("F",11)]
```

Second, instead of directly using the `Fork` data constructor to build a new queue, we need a smart constructor function `fork` which takes an element and a priority, two queues and return a new queue that has two subtrees and with the order value automatically updated (note that the new queue should obey the property of "leftist heap").

```haskell
Fork :: Order -> a -> p -> PQueue a p -> PQueue a p -> PQueue a p

fork :: a -> p -> PQueue a p -> PQueue a p -> PQueue a p
fork = undefined
```

Expected Results

```haskell
sampleTree1' :: PQueue String Int
sampleTree1' = fork "A" 4 (fork "B" 5 (fork "C" 6 Null Null) Null)
                          (fork "D" 8 Null Null)

sampleTree2' :: PQueue String Int
sampleTree2' = fork "A" 1 (fork "B" 3 (fork "C" 4 Null Null)
                                      (fork "D" 6 Null Null))
                          (fork "E" 10 (fork "F" 11 Null Null) Null)
```

```
Main> sampleTree1 == sampleTree1'
True
Main> sampleTree2 == sampleTree2'
True
```

Then, we need to define a `mergeQ` function to merge two leftist heaps and create a new leftist heap.

```
mergeQ :: Ord p => PQueue a p -> PQueue a p -> PQueue a p
mergeQ = undefined
```

Expected Results

```
foo1 = fork "A" 10 Null Null
foo2 = fork "B" 100 Null Null
foo3 = fork "C" 1 Null Null

Main> flattenQ $ mergeQ sampleTree1 sampleTree2
[("A",1),("B",3),("C",4),("A",4),("B",5),("D",6),("C",6),("D",8),("E",10),("F",11)]
Main> mergeQ foo1 foo2
Fork 1 "A" 10 (Fork 1 "B" 100 Null Null) Null
Main> mergeQ (mergeQ foo1 foo2) foo3
Fork 1 "C" 1 (Fork 1 "A" 10 (Fork 1 "B" 100 Null Null) Null) Null
Main> mergeQ (mergeQ foo1 foo3) foo2
Fork 2 "C" 1 (Fork 1 "A" 10 Null Null) (Fork 1 "B" 100 Null Null)
```

Finally, we can implement our two interface functions using the functions we defined above:

```
insert :: Ord p => a -> p -> PQueue a p -> PQueue a p
insert = undefined

delete :: Ord p => PQueue a p -> ((a, p), PQueue a p)
delete = undefined
```

Please finish the implementation of `flattenQ`, `fork`, `mergeQ`, `insert` and `delete` according to the properties we specified.