Introduction to Programming: Practice Assignment

Set: November 19, 2026 **Due:** December 4, 2026, 2359 hrs.

General instructions

- As far as possible, use only the `Prelude`, `Data.Char`, `System.IO`, `System.Exit`, `Control.Monad`, `Text.Read` (for `readMaybe`), and your own modules. Do *not* use `Data.Map`, `Data.List` or any other library module.
- The Stack and Queue modules from the lectures are referred to as `Stack` and `Queue`. Where a question says "import `Stack`", use only the functions it exports: `empty`, `isEmpty`, `push` and `pop :: Stack a → (a, Stack a)`.
- The programs in Part B may be tested by *input redirection*, as in `./RunningSum < test1.txt`, and the output will be compared with the expected output. Every program must stop cleanly at the end of input (use `isEOF`).
- Try to attain the stated running times. Aim for correct solutions first even if inefficient, and then try to achieve the stated asymptotic complexity.
- *You are not required to submit this assignment. It is for learning and practice. But if you attempt problems worth at least 40 marks and submit before the due date, you will get additional marks (upto 5). You must submit each answer in a separate file, as specified. The programs will be tested by scripts, so if there are compilation errors, you will get a zero for that question.*

Part A: Data structures as modules

1. **A stack with a minimum** (10 marks). File `MinStack.hs`.

   Implement a stack that, in addition to the usual operations, returns its smallest element in $O(1)$ steps.

```
module MinStack(MinStack, empty, isEmpty, push, pop, top, minElem) where

empty   :: MinStack a
isEmpty :: MinStack a → Bool
push    :: Ord a ⇒ a → MinStack a → MinStack a
pop     :: MinStack a → (a, MinStack a)
top     :: MinStack a → a
minElem :: MinStack a → a
```

   - Every operation must take $O(1)$ steps. `pop`, `top` and `minElem` call `error` on an empty stack.
   - Define `instance Show a ⇒ Show (MinStack a)` that shows the elements from top to bottom as a list. For example, `show (push 3 (push 1 (push 4 empty)))` is `"[3,1,4]"`. It must not reveal anything else about your representation.
   - *Hint:* along with each element, store the minimum of the elements at or below it.

2. **A queue built from two stacks** (10 marks). File `StackQueue.hs`.

   Implement a queue using *two* values of type `Stack a`, importing the lecture's `Stack` module.

```
module StackQueue(Queue, empty, isEmpty, enqueue, dequeue, size, toList) where
import Stack

enqueue :: a → Queue a → Queue a
dequeue :: Queue a → (a, Queue a)
size    :: Queue a → Int
toList  :: Queue a → [a]
```

Here `enqueue` adds at the rear, `dequeue` removes from the front (and calls `error` on an empty queue), and `toList` lists the elements from front to rear.

- You may use only the functions exported by `Stack`: you cannot pattern-match on stacks. Note the name clash: both modules define `empty` and `isEmpty`. Use `import qualified Stack as S` and write `S.empty`, `S.push`, and so on.
- `enqueue` must take $O(1)$ steps, and `dequeue` $O(1)$ steps amortised over a sequence of operations. `size` must take $O(1)$ steps, so store the size in the queue.
- In a comment at the top of the file, explain in three or four sentences why any sequence of n `enqueue` and `dequeue` operations, starting from `empty`, takes $O(n)$ steps in total.

3. **A dictionary as a binary search tree** (25 marks). File `BSTMap.hs`.

   A *map* stores key–value pairs, at most one pair for each key. Implement it as a binary search tree ordered on the keys.

```
module BSTMap(Map, empty, null, size, insert, insertWith, lookup, member,
              delete, fromList, toList, lookupLE, rangeList) where
import Prelude hiding (lookup, null)

data Map k v = Tip | Bin (Map k v) k v (Map k v)
```

The `import` line hides the Prelude's own `lookup` and `null`, so that yours can be defined. Users of your module should write `import qualified BSTMap as M` and use `M.insert`, `M.lookup`, and so on. Define these functions:

```
empty      :: Map k v
null       :: Map k v → Bool
size       :: Map k v → Int
insert     :: Ord k ⇒ k → v → Map k v → Map k v
lookup     :: Ord k ⇒ k → Map k v → Maybe v
member     :: Ord k ⇒ k → Map k v → Bool
insertWith :: Ord k ⇒ (v → v → v) → k → v → Map k v → Map k v
delete     :: Ord k ⇒ k → Map k v → Map k v
toList     :: Map k v → [(k, v)]
fromList   :: Ord k ⇒ [(k, v)] → Map k v
```

```
lookupLE   :: Ord k ⇒ k → Map k v → Maybe (k, v)
rangeList  :: Ord k ⇒ k → k → Map k v → [(k, v)]
```

Below, h is the height of the tree and n its size.

(a) (4 marks) `empty`, `null`, `size` and `insert`. If the key is already present, `insert` replaces its value.

(b) (3 marks) `lookup` and `member`, in $O(h)$ steps.

(c) (3 marks) `insertWith f k new m`: if `k` is absent, this behaves like `insert`. If `k` has value `old`, its new value is `f new old`. For example, `insertWith (+) "cat" 1 m` increments the count of `"cat"`.

(d) (4 marks) `delete`, following the lecture's deletion from a BST. Deleting an absent key leaves the map unchanged.

(e) (4 marks) `toList`, which lists the pairs in increasing order of keys in $O(n)$ steps (beware of `++`), and `fromList`. If a key occurs several times in the input of `fromList`, the *last* occurrence wins. Note that `foldr insert` from the lectures does not achieve this.

(f) (3 marks) `lookupLE k m` returns the pair with the largest key that is $\leqslant$ `k`, in $O(h)$ steps. For example, if the keys are `10`, `20` and `30`, then `lookupLE 25` returns the pair for `20`, and `lookupLE 5` returns `Nothing`.

(g) (4 marks) `rangeList lo hi m` lists, in increasing order of keys, the pairs whose keys lie between `lo` and `hi` (inclusive). It must not visit any subtree that cannot contain such a key, so it takes $O(h+r)$ steps, where r is the number of pairs returned.

Also define `instance (Show k, Show v) ⇒ Show (Map k v)`, so that `show (fromList [(2,'b'),(1,'a')])` is `"fromList [(1,'a'),(2,'b')]"`. This part has no separate marks, but it will be used in testing.

Part B: Input and output

4. **Running sum, with undo** (15 marks). File `RunningSum.hs`.

   (a) (7 marks) Read the input one line at a time until the end of input. If the line is an integer, print it back. If the line is `s`, print the sum of all the integers read so far, in the form `sum = 17`.

   (b) (8 marks) Extend the program with these commands, handled in the same file:

   - `u` (undo) removes the most recently entered number that has not already been undone, and prints `undo 7`. If there is no such number, it prints `nothing to undo`. Repeated `u` commands undo numbers further back.
   - `q` stops the program immediately, printing `bye`. The program also stops, silently, at the end of input.
   - Blank lines are ignored. Any other line prints `invalid input`.

   The command `s` must take $O(1)$ steps, however many numbers have been entered. Keep the entered numbers on a `Stack` and the running sum in a separate argument of your loop. *Hint:* `readMaybe line :: Maybe Int` (from `Text.Read`) returns `Nothing` if `line` is not an integer.

Input:

```
5
7
s
-2
s
u
u
s

hello
q
9
```

Output:

```
5
7
sum = 12
-2
sum = 10
undo -2
undo 7
sum = 5
invalid input
bye
```

5. **Checking brackets** (15 marks). File `Brackets.hs`.

   Read lines until the end of input. For each line, check whether its brackets `()`, `[]` and `{}` are balanced, ignoring all other characters, and print one line:

   - `ok` if the brackets are balanced (this includes lines with no brackets);
   - `mismatch at p` if the first problem is a closing bracket at position `p` (counting from 1) that does not match the most recent unclosed opening bracket, or that has no opening bracket at all;
   - `unclosed at p` if the line ends with some opening brackets unclosed, where `p` is the position of the *most recent* unclosed one.

   After the last line, print `k of n lines balanced`. Use the lecture's `Stack`, storing each opening bracket together with its position. Each line must be processed in time proportional to its length.

Input:

```
f(a[i]) + {x}
(a + b]
((x)
{[}]
)
no brackets here
```

Output:

```
ok
mismatch at 7
unclosed at 1
mismatch at 3
mismatch at 1
ok
2 of 6 lines balanced
```

6. **Word frequencies** (15 marks). File `WordFreq.hs`. Import `BSTMap` from Question 3.

   Read text until the end of input. Treat every character that is not a letter as a space, and convert letters to lower case, so that `The`, `the,` and `THE` are the same word.

   (a) (9 marks) Print each word and its number of occurrences, one per line, in alphabetical order. Then print a summary line, as in the example. Update a `Map String Int` with `insertWith` as each line is read; do not store the lines themselves.

   (b) (6 marks) Finally, print `top:` followed by the three most frequent words (fewer, if there are fewer distinct words). Break ties alphabetically. Sort using a merge sort that you write yourself, on the list produced by `toList`.

Input:

```
The cat and the hat.
A cat, a bat!
```

Output:

```
a 2
and 1
bat 1
cat 2
hat 1
the 2
6 distinct words, 9 in total
top: a cat the
```

7. **A service counter** (15 marks). File `Counter.hs`. Import `StackQueue` from Question 2.

   Simulate a queue at a service counter. Read commands, one per line, until the end of input or `quit`:

   - `join NAME` adds `NAME` at the rear and prints `NAME joined, k waiting`, where `k` is the new length of the queue;
   - `serve` removes the person at the front and prints `serving NAME`, or prints `no one to serve` if the queue is empty;
   - `show` prints `waiting:` followed by the names from front to rear, separated by single spaces, or `waiting: none`;
   - `quit`, or the end of input, prints `closing, k still waiting` and stops;
   - any other line prints `unknown command`.

   Hint: pattern-match on `words line`, as in `["join", name]` and `["serve"]`.

Input:

```
join Asha
join Ravi
show
serve
join Meena
serve
serve
serve
dance
join Kumar
```

Output:

```
Asha joined, 1 waiting
Ravi joined, 2 waiting
waiting: Asha Ravi
serving Asha
Meena joined, 2 waiting
serving Ravi
serving Meena
no one to serve
unknown command
Kumar joined, 1 waiting
closing, 1 still waiting
```

8. **Guess the number** (15 marks). File `Guess.hs`

   Pick a secret number with `randomRIO (1, 100)` (from `System.Random`; you may need to run `cabal install --lib random` first). Repeatedly prompt `Your guess:` (use `hFlush stdout`), and reply `too low` or `too high`, until the guess is right. Then print `correct, in k guesses`. Lines that are not numbers are ignored and do not count as guesses. As an extension, after each game ask `Play again? (y/n)`, and at the end print the best score.