Working on your assignment

You can work on this assignment on your own computer or the lab machines. It is important that you continually back up your assignment files onto your own machine, external drives, and in the cloud. You are responsible for your assignment files to remain private to you, and not accessible by others.

You are encouraged to submit your assignment on Ed while you are in the process of completing it. By submitting you will obtain some feedback of your progress on the sample test cases provided.

If you have any questions about any C functions, then refer to the corresponding **`man`** pages. You can ask questions about this assignment on Ed. As with any assignment, make sure that your work is your own, and that you do not share your code or solutions with other students.

Where to start

Begin with simple entries and their operations.

- confirm you can process the line of input and recognise commands, integers (of multiple digits)
- implement the SET operation
- implement the simple entry operations and also calculations (min,max,sum,len)
- check that you can retain the value of a new entry and query its value, destroy it, and check that it no longer exists

Next, approach snapshots and confirm your test data and operations of input meet your expectation.

Next, approach general entries. This requires more care with forward/backward references. Look to linked list data structures and traversals for hints.

Simple entry (strictly integer values)

The simple entry is a key value store, where only integers are used as values. The following defines and initialises an entry with key identifier `a` with values:

```
> SET a 1 2 3
ok

> GET a
[1 2 3]
```

The values can be updated with other commands.

```
> SET a 1 2 3
ok

> PUSH a 5
ok

> GET a
[5 1 2 3]
```

```
> APPEND a 7
ok

> GET a
[5 1 2 3 7]

> SET a 9 8 7
ok

> GET a
[9 8 7]
```

An entry is termed *simple* if it's values are only integers. It is otherwise *general* and described later.

Managing states of the database

What is state?

The state of the database is comprised of all the keys, and their associated values at a given point in time. For example, when creating multiple keys, and their values, their state is preserved in memory.

What are snapshots?

A user may choose to save the current state of the database by calling SNAPSHOT. By doing so, a copy of the current state is saved and can be later referred to by it's snapshot number.

```
> SET a 1 2 3
ok

> SNAPSHOT
saved as snapshot 1
```

At a later time, the user may choose to replace the current state with a snapshot created at an earlier time. This is done using the CHECKOUT command. This does not affect other snapshots in memory.

```
> SET a 1 2 3
ok

> SNAPSHOT
saved as snapshot 1

> SET a 4 5 6
ok

> SNAPSHOT
saved as snapshot 2
```

```
> SET a 7 8 9
ok

> CHECKOUT 1
ok

> GET a
[ 1 2 3 ]

> CHECKOUT 2
ok

> GET a
[ 4 5 6 ]
```

Alternatively, the user may choose to restore the state by loading a particular snapshot and discard all changes or snapshots made since that time. This is the ROLLBACK command.

```
> SET a 1 2 3
ok

> SNAPSHOT
saved as snapshot 1

> SET a 4 5 6
ok

> SNAPSHOT
saved as snapshot 2

> SET a 7 8 9
ok

> ROLLBACK 1
ok

> GET a
[ 1 2 3 ]

> CHECKOUT 2
no such snapshot
```

Actions on current state with regards to snapshot

Snapshots and the current state are mutually disjoint. Actions on the current state should not influence any other snapshots. Note, for PURGE, it is not an action **ONLY** on the current state, see notes on PURGE.

General entry (mixed integer and key values)

We recommend completing all commands with the simple entry before beginning this part of the assignment

A general entry contains at least one value that is a key to another entry in the database.

```
> SET a 1 2
ok

> SET b 4 a 6
ok

> LIST ENTRIES
B [ 4 a 6 ]
A [ 1 2 ]

> SUM a
3

> SUM b
13
```

An entry cannot store a value being it's own key, or to a key that does not exist in the current state.

```
> SET a a
not permitted

> SET b z
no such key
```

When an entry refers to another key, a bidirectional relationship of forward and backward references between these entries need to be maintained.

- Entry X has a forward reference of an entry Y if X contains the key of Y as one of it's values
- Entry X has a backward reference of an entry Y if Y contains the key of X as one of it's values

To understand forward and backward references, consider the following example.

```
> SET c 7
ok

> SET b 3 5 c
ok

> SET a 1 2 b
```

```
ok

> SUM c
7

> SUM b
15

> SUM a
18
```

- a has a forward reference to b
- b has a forward reference to c
- c has no forward reference
- c has a backward reference to b
- b has a backward reference to a
- a has no backward references

Forward and backward references of a entry, and including the subsequent entries within, can be queried using the FORWARD and BACKWARD commands.

```
> SET c 7
ok

> SET b 3 5 c
ok

> SET a 1 2 b
ok

> FORWARD a
b, c

> FORWARD b
c

> FORWARD c
nil

> BACKWARD a
nil

> BACKWARD b
```

```
a

> BACKWARD c
a, b
```

The output of these commands is a search for all forward or backward references related to this entry. The output presents the unique keys in lexicographical sorted order.

Understanding a valid state

The current state is valid if and only if every general entry has a corresponding key exist.

Operations affecting the valid state include the removal of forward or backward references. DEL and PURGE are described using the following example.

Suppose there are forward and backward references created.

```
> SET c 7
ok

> SET b 3 5 c
ok

> SET a 1 2 b
ok
```

The state of the database above is valid. Consider the order of commands in the following:

```
> DEL c
not permitted

> DEL b
not permitted

> DEL a
ok

> DEL c
not permitted

> DEL b
ok

> DEL c
ok
```

If c were to be deleted first, the state of the database would be inconsistent, due to c being referred to by b. c has back references.

Similarly, if b were to be deleted, the state of the database would be inconsistent, due to it being referenced by a. b has back references.

The operation is performed if and only if the resulting state is valid.

If DEL a were to be performed, a has no backward references, therefore the state after removal of a would be valid and can proceed. After this, b no longer has backward reference to a and DEL b can proceed.

The DEL command operates on the current state.

PURGE operation

The PURGE operation is performed if and only if the resulting states of all snapshots and also the current state are valid.

If the key does not exist in the state or the snapshot, the operation can proceed as the state is valid. PURGE does not need to report keys that do not exist.

For example, consider the empty state

```
> DEL g
no such key

> PURGE g
ok
```

Snapshots are always created as a valid state, these can be broken with PURGE. Consider a more elaborate scenario where there are 2 snapshots and the current state.

PURGE scenario

```
> SET c 7
ok

> SET b 3 5 c
ok

> SET a 1 2 b
ok

> SNAPSHOT
saved as snapshot 1

> DEL a
ok

> SNAPSHOT
saved as snapshot 2
```

```
> SET b 1 2 3
ok

> PURGE b
not permitted

> PURGE a
ok
```

PURGE b removes all keys b only if each state remains valid. Consider from the scenario:

- if removal of b from current state. b's entry is simple, no backward references, thus the state is valid.
- if removal of b from snapshot 2. b's entry is simple, no backward references, thus the state is valid.
- if removal of b from snapshot 1, b's entry has a backward reference a. a cannot be resolved after removal and the state would become invalid.

Therefore, PURGE b will not modify any state.

PURGE a removes all keys a only if each state remains valid. Consider from the scenario:

- if removal of a from current state. a does not exist, it is ignored.
- if removal of a from snapshot 2. a does not exist, it is ignored.
- if removal of a from snapshot 1, a has no backward references. It can be removed and the state remains valid.

PURGE a will remove keys a from all snapshots and current state if it exists.

A note about removal of keys from an index

A general entry can remove a value, being a key, from it's list of values. If the key exists, and is removed, then the state remains valid.

PLUCK, POP are such operations. It is also possible to use SET to remove a value from an entry by redefining its values.

Care should be taken to update the backward and forward references such that subsequent operations following PLUCK or POP are consistent with the expected behaviour.

Implementation details

Write a program in C that implements YmirDB as shown in the examples. You can assume that our test cases will contain only valid input commands and not cause any integer overflows.

Keys are case sensitive and do not contain spaces. Keys are alphanumeric and must begin with an alphabetical character [a-z][A-Z]. The key length is no greater than 15 characters.

Commands are case insensitive.

Entry values are indexed from 1.

Snapshots are indexed from 1 and are unique for the lifetime of the program.

Keys, entries and snapshots are to be displayed in the order from most recently added to least recently added.

A snapshot's keys and values should not have allocated memory associated with other snapshots.

Output for commands forward or backward references present the sorted lexicographical order by key and unique keys only.

Test cases will not include self referencing or reference cycles.

Your program must be contained in ymirdb.c and ymirdb.h and produce no errors when built and run on Ed C compiler. Reading from standard input and writing to standard output.

Your program output must match the exact output format shown in the examples and on Ed. You are encouraged to submit your assignment while you are working on it, so you can obtain feedback.

The contents of an example header file ymirdb.h are shown below. You may modify, or make your own.

No external code outside the standard C library functions should be required.

```c
#ifndef YMIRDB_H
#define YMIRDB_H

#define MAX_KEY 16
#define MAX_LINE 1024

#include <stddef.h>
#include <sys/types.h>

enum item_type {
    INTEGER=0,
    ENTRY=1
};

typedef struct element element;
typedef struct entry entry;
typedef struct snapshot snapshot;
```

```
struct element {
  enum item_type type;
  union {
    int value;
    struct entry *entry;
  };
};

struct entry {
  char key[MAX_KEY];
  char is_simple;
  element * values;
  size_t length;

  entry* next;
  entry* prev;

  size_t forward_size;
  size_t forward_max;
  entry** forward;  // this entry depends on these

  size_t backward_size;
  size_t backward_max;
  entry** backward; // these entries depend on this
};

struct snapshot {
  int id;
  entry* entries;
  snapshot* next;
  snapshot* prev;
};

#endif
```

In order to obtain full marks, your program must free all of the dynamic memory it allocates. This will be automatically checked using address sanitizer. If your program produces the correct output for a given test case and does not free all memory it allocates, then it will not pass the given test case.

Commands

Your program should implement the following commands, look at the examples to see how they work.

- If a <key> does not exist in the current state, output: no such key
- If a <snapshot> does not exist in the database, output: no such snapshot
- If an <index> does not exist in an entry, output: index out of range
- If an <entry> cannot be removed, output: not permitted

```
BYE   clear database and exit
HELP  display this help message

LIST KEYS       displays all keys in current state
LIST ENTRIES    displays all entries in current state
LIST SNAPSHOTS  displays all snapshots in the database

GET <key>     displays entry values
DEL <key>     deletes entry from current state
PURGE <key>   deletes entry from current state and snapshots

SET <key> <value ...>     sets entry values
PUSH <key> <value ...>    pushes values to the front
APPEND <key> <value ...>  appends values to the back

PICK <key> <index>      displays value at index
PLUCK <key> <index>     displays and removes value at index
POP <key>               displays and removes the front value

DROP <id>       deletes snapshot
ROLLBACK <id>   restores to snapshot and deletes newer snapshots
CHECKOUT <id>   replaces current state with a copy of snapshot
SNAPSHOT        saves the current state as a snapshot

MIN <key>   displays minimum value
MAX <key>   displays maximum value
SUM <key>   displays sum of values
LEN <key>   displays number of values

REV <key>   reverses order of values (simple entry only)
UNIQ <key>  removes repeated adjacent values (simple entry only)
SORT <key>  sorts values in ascending order (simple entry only)

FORWARD <key> lists all the forward references of this key
BACKWARD <key> lists all the backward references of this key
TYPE <key> displays if the entry of this key is simple or general
```

Examples

```
> LIST KEYS
no keys

> LIST ENTRIES
no entries

> LIST SNAPSHOTS
no snapshots

> SET a 1
ok

> GET a
[1]

> POP a
1

> GET a
[]

> POP a
nil

> PUSH a 2 1
ok

> GET a
[1 2]

> APPEND a 3 4
ok

> GET a
[1 2 3 4]

> DEL a
ok

> DEL a
no such key

> BYE
bye
```

```
> SET a 1
ok

> SET b 2 3
ok

> LIST KEYS
b
a

> LIST ENTRIES
b [2 3]
a [1]

> LIST SNAPSHOTS
no snapshots

> PICK a 0
index out of range

> PICK b 1
2

> GET b
[2 3]

> PLUCK b 2
3

> GET b
[2]

> DEL b
ok

> GET b
no such key

> PURGE b
ok

> BYE
bye
```

```
> DROP 1
no such snapshot

> ROLLBACK 1
no such snapshot

> SET a 1 2
ok

> SET b 3 4
ok

> LIST ENTRIES
b [3 4]
a [1 2]

> SNAPSHOT
saved as snapshot 1

> SET c 5 6
ok

> LIST ENTRIES
c [5 6]
b [3 4]
a [1 2]

> SNAPSHOT
saved as snapshot 2

> PURGE b
ok

> ROLLBACK 1
ok

> CHECKOUT 2
no such snapshot

> LIST ENTRIES
a [1 2]

> LIST SNAPSHOTS
1

> BYE
bye
```

```
> SET a 1 4 2 3 4 2
ok

> SNAPSHOT
saved as snapshot 1

> MIN a
1

> MAX a
4

> SUM a
16

> LEN a
6

> REV a
ok

> GET a
[2 4 3 2 4 1]

> SORT a
ok

> GET a
[1 2 2 3 4 4]

> UNIQ a
ok

> GET a
[1 2 3 4]

> CHECKOUT 1
ok

> LIST SNAPSHOTS
1

> LIST ENTRIES
a [1 4 2 3 4 2]

> BYE
bye
```

```
> SET a 1 2 3
ok

> SET b 4 5 6
ok

> PUSH a b
ok

> LIST ENTRIES
b [4 5 6]
a [b 1 2 3]

> GET a
[b 1 2 3]

> GET b
[4 5 6]

> TYPE b
simple

> TYPE a
general

> FORWARD a
b

> FORWARD b
nil

> BACKWARD a
nil

> BACKWARD b
a

> DEL b
not permitted

> DEL a
ok

> BYE
bye
```

```
> SET a 1 2 3
ok

> SET b 4 5 6
ok

> APPEND a b
ok

> GET a
[1 2 3 b]

> MAX a
6

> SUM a
21

> LEN a
6

> PICK a 4
b

> PLUCK a 3
3

> GET a
[1 2 b]

> PLUCK a 3
b

> GET a
[1 2]

> PUSH a b
ok

> GET a
[b 1 2]

> POP a
b

> BYE
bye
```

Submission details

You are encouraged to submit multiple times, but only your last submission will be marked.

Writing your own test cases

We have provided you with some test cases but these do not test all the functionality described in the assignment. It is important that you thoroughly test your code by writing your own test cases.

You should place all of your test cases in the tests/ directory. Ensure that each test case has the .in input file along with a corresponding .out output file. We recommend that the names of your test cases are descriptive so that you know what each is testing, e.g. get-set.in and sort-uniq.in

Restrictions

If your program does not adhere to these restrictions, your submission will receive 0.

- No Variable Length Arrays (VLAs)
- No Excessive CPU usage (algorithms must be better than $\Theta(n^2)$ average quadratic time)

Marking

A grade for this assignment is made where there is a submission that compiles and the oral examination has been completed.

15 **marks** are assigned based on automatic tests for the *correctness* of your program.
This component will also use hidden test cases to cover the assignment description.

5 **marks** are based on your submission of test cases, which considers coverage of input scenarios.

10 **marks** are based on your oral examination and code quality.

Additionally marks will be deducted if:

- your program has memory leaks.
- excessive memory is used, e.g preallocation or over allocation of what is required.
- your program avoids *malloc* family of C functions to instantiate memory dynamically.
- uses unjustifiable magic numbers.
- uses files, or mapped memory.

Warning: Any attempts to deceive the automatic marking will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not properly follow the assignment description, or your code is unnecessarily or deliberately obfuscated.