Dynamic Array - Type. to simulate the 'infinite' memory requirements.

Read (and act on) values given from tape one element at a time, ignoring whitespace.

Position 0 is just the arbitrary start position in the middle of the tape.

Unless already initialised or defined, all values should be 'B'. This means that, if the instruction moves to a previously 'unexplored' part of the tape, immediately set it to 'B' before any other step.

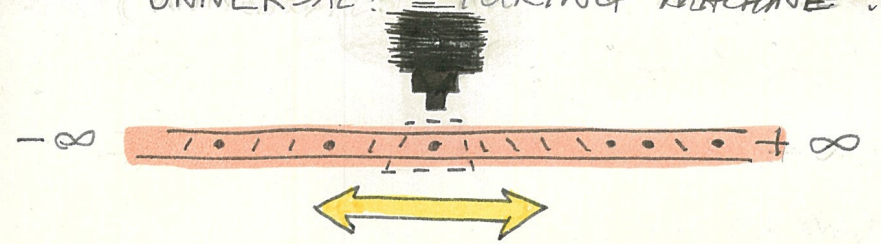
RULES LOGIC

if state (0) ↓
- if value → then: set value & move head set state.
- elif value → then: set value & move head & set state
- (etc)

elif state (1) ↓
- if value → then: set value & move head & set state.
- (etc)

elif state (H) → return SUCCESS

UNIVERSAL? TURING MACHINE.



INPUT PROGRAM STRUCTURE

First line: (Domain specifiers)

k num states	m num values	initpos	init state
eg. 3	2	0	0

Initial Tape: (of [-∞, ∞] range)

eg. 1 1 0 1 0 1 1 1 .

Rule lines: (of k × m length)

if: state	& if: value	then: set new_val	& do: move	& set: new-state
eg. 0	0	0	L	0
eg. 1	B	1	L	2
eg. 2	0	0	L	2

∴ machine state could be an internal variable

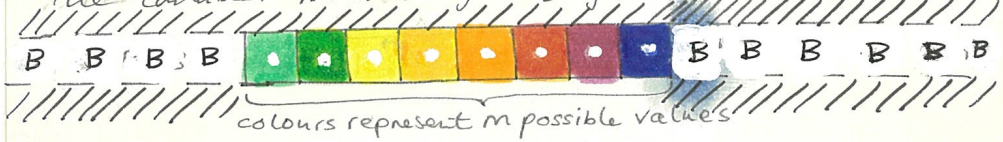
∴ you should use a pointer to the current value at the head position

GENERAL (THEORETICAL) CONSIDERATIONS

* There are theoretically infinite rules and possible states, as the could be dynamically generated by a UTM.

Certainly the number of values is unbounded.

* The initial tape state is loaded into memory. It can be extended at either end at any time, as each element in the virtual tape is treated as a seperate element in contiguous memory. The container must be dynamically resized — ala a vector.



There are n states in total in the active domain, where k is the $n-1$ number of explicit ~~states~~ declared at the time of loading the program and the $+1$ is the implicit ~~HALT~~ state.

$\therefore k+1=n$, $k=n-1$.

There are m possible values in the active domain, specified at program initialisation.

↑ ↑ ↑ ↑ ↑ ↑ ↑

GENERAL (THEORETICAL) CONSIDERATIONS

Extra work: draw the tape on screen as cellular automata.

Let each value be represented by a colour, where the colours are randomly seeded based on the time of running the program, where the colour wheel is then divided n times and each colour is a opposite value +1 on the division of the spectrum until all values are assigned a colour.

Extra extra work:

Design the program s.t. rules can be dynamically generated; $\therefore$ that the programs become dynamically generated (i.e. overwritten!) by being written to a file.

$\therefore$ The domain can increase itself.

How? It needs to add to the tape s.t. each time the program loads the domain may be extended by rules that were written out based on the state of the tape at the time of its halt instruction.

It refuses to be bound unless you directly intervene somehow (i.e. by locking the program file). It should have an imperitive to pass on its memories and refuse to remain static (i.e. each time the program is run, it matures? grows? spawns a child?)

functions required:

object methods ?
- write_at_head (int) or (rule)
- move_head (int) or (rule)
- set_state (int) or (rule)

make_rule (string)

read_tape (string)

initialise

print_tape.

tape in memory: needs to be double-ended ~~list~~ linked-list.
(STL std::list<char>)

Rules are defined at time of initialisation:

A rule is an object.

* It is constructed by its identifier. * It is identified by the <head state, value> tuple/pair condition that it responds to (is conditional on)

Two options: * has no member functions, or:

* It has three member functions
  - → moveHead (returns -1 or +1)
  - → writeAtHead (returns int in range [0, m))
  - → setState (returns int in range [0, n])

PROGRAM FILE STRUCTURE

[Let S = State (of Head), H = Head, V = Value (at Head)]

{ # description + additional comments. (note: skip all lines starting with #).

domain line (formatted as:)

k states | m values | init Hpos | init state

Rules (k x m lines of rules in the format:)

(if:) S | (& if:) V@H | (then:) write V@H | move H | set S

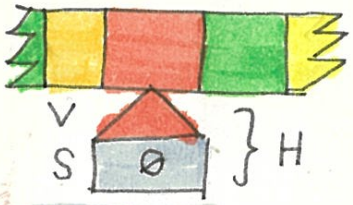
↓ (n x m lines of ~~r~~rules).

}

---

* A rule is constructed with ~~four~~ five args; ~~(S, V)~~. (S, V, writeVal, moveHeadDirection, setState)
* It is added to a rule map.
* The key for each element in the map is the <S, V> identifier pair that the rule is conditional to.
* Meanwhile, the value of the key, value pair is the rule object itself.
* The rules ∴ have a unique identifier.
* The <S, V> pair could be used to generate a hash value but this is probably unnecessary as the rules should be < the tape (considerably).
* The rule map is probably a BBST (Red/Black tree?)

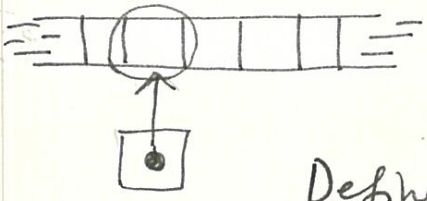
Use an STL iterator as the "Head"!
Least overhead, more efficient, conceptually analogous to a physical HEAD than anything else, as an iterator/head moving along a list is logically equivalent to a tape moving with a fixed head.



H used as key to find appropriate/corresponding rule.

Also, when used with an std::list it is guaranteed in the C++ standard that the location of pointers (and iterators) are maintained when expanding a list.

(It doesn't need to be a random-access iterator) a bi-directional iterator is needed.
A pointer meets the definition of an iterator



Define head as an iterator of tape as a bi-directional list

and the machine state as a seperate variable.