iptable

A Lua binding to `iptable.c`, a thin wrapper around *radix.c* from the *FreeBsd* project which implements a radix tree with one-way branching removed (a level collapsed trie).

`iptable` provides some convenience functions to work with ip addresses and subnets (in CIDR notation) as well as a longest prefix matching (Lua) table. It handles both ipv4 and ipv6 addresses and/or subnets transparently.

Throughout the documentation `prefix` refers to a string that represents either an ipv4 or ipv6 address or network in CIDR notation.

Requirements and limitations

`iptable` is developed on a linux machine and support for other OS's is non-existent at this point. The following is currently used to build iptable:

```
- Lua 5.4.6  Copyright (C) 1994-2023 Lua.org, PUC-Rio
- Ubuntu 24.04 noble
- gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
- GNU Make 4.3
```

`lua_iptable.c` uses Lua 5.3's C-API, so Lua >= 5.3 is required. Additionally for testing and documentation, the following is used:

```
- valgrind-3.22.0
- busted 2.2.0
- pandoc 3.1.3
- pandoc-imagine
```

If you have trouble getting pandoc_imagine to work, just:

- copy file to `~/bin` (or any other directory on your `$PATH`).
- make it executable (`chmod u+x ~/bin/pandoc_imagine.py`)
- and the run `make readme`

Pandoc expects its filters to be executable and searches `$PATH`.

Installation

Make.

```
cd ~/installs
git clone https://github.com/hertogp/iptable.git
cd iptable
make test
sudo -H make install
# or
make local_install    # simply copies to ~/.luarocks/lib/lua/5.4
```

luarocks

- todo.

C-only

Just copy the files `src/iptable.{h,c}`, `src/radix.{h.c}` and `src/debug.h` to your project. additional documentation in the doc directory. Alternatively, the Makefile has a `c_test` and a `c_lib` target to test and to build `build/libiptable.so`.

Usage

An iptable.new() yields a Lua table with modified indexing behaviour:

- the table utilizes 2 separate radix trees for ipv4 and ipv6 respectively
- the table handles both ipv4 or ipv6 keys transparently
- if a key is not an ipv4 or ipv6 subnet/address *string* it is *ignored*
- ipv4 and ipv6 subnets are always in CIDR notation: address/len
- storing data always uses a subnet-key (address/mlen)
- when storing data, missing masks default to the AF's maximum mask
- when storing data, masks are always applied first to the key
- storing data is always based on an exact match which includes the mask
- retrieving data with a subnet-key, uses an exact match
- retrieving data with an address-key, uses a longest prefix match
- the `iptable.size(pfx)` function uses Lua arithmatic, hence the float
- `mlen == -1` signals the absence of a max
- it is safe to delete entries while iterating across the table

Example usage:

```
ipt = require"iptable".new()

ipt["10.10.10.0/24"] = {seen=0}       -- store anything in the table
ipt["10.10.10.10"] = false            -- stores to ipt["10.10.10.10/32"]
ipt["11.11.11.11/24"] = true          -- stores to ipt["11.11.11.0/24"]
ipt["acdc:1976::/32"] = "Jailbreak"   -- separate class (radix tree) of its own
ipt[1] = 42                           -- ignored: ipt[1] -> nil
iptable.error                         -- "wrong type of argument"
#ipt                                  -- 4 entries in total
ipt.counts()                          -- 3  1  (ipv4 and ipv6 counts)

for k,v in pairs(ipt) do              -- 10.10.10.0/24    table 0x...
    print(k,v)                        -- 10.10.10.10/32   false
                                      -- 11.11.11.0/24    true
end                                   -- acdc:1976::/32   Jailbreak
```

Quick reference

```
iptable = require "iptable"

-- Module constants

iptable.AF_INET  --  2
iptable.AF_INET6 -- 10

-- Module functions

prefix = "10.10.10.0/24"                          -- ipv4/6 address or subnet

addr,   mlen, af = iptable.address(prefix)        -- 10.10.10.0      24  2
bcast,  mlen, af = iptable.broadcast(prefix)      -- 10.10.10.255    24  2
netw,   mlen, af = iptable.network(prefix)        -- 10.10.10.0      24  2
neigb,  mlen, af = iptable.neighbor(prefix)       -- 10.10.11.0      24  2
inv,    mlen, af = iptable.invert(prefix)         -- 245.245.245.255 24  2
rev,    mlen, af = iptable.reverse(prefix)        -- 0.10.10.10      24  2
addr,   mlen, af = iptable.longhand("2001::")     -- 2001:0000:...   -1  10
addr,   mlen, af = iptable.offset(prefix, -6)     -- 10.10.9.250     24  2
p1, p2, mlen, af = iptable.split(prefix)          -- 10.10.10.0 10.10.10.128 25 2

mask = iptable.mask(iptable.AF_INET, 24)          -- 255.255.255.0
size = iptable.size(prefix)                       -- 256.0
ptr  = iptable.dnsptr(prefix)                     -- 0.10.10.10.in-addr.arpa.
ptr  = iptable.dnsptr(prefix, true)               -- 10.10.10.in-addr.arpa.

binkey = iptable.tobin("255.255.255.0")           -- byte string 05:ff:ff:ff:00
prefix = iptable.tostr(binkey)                    -- 255.255.255.0
msklen = iptable.masklen(binkey)                  -- 24

ipt    = iptable.new()                            -- longest prefix match table

for host in iptable.hosts(prefix[, true]) do      -- iterate across hosts in prefix
    print(host)                                   -- optionally include netw/bcast
end

for pfx in iptable.subnets(prefix, 26) do         -- iterate prefix's subnets
    print(pfx)                                    -- new prefix len is optional
end                                               -- and defaults to 1 bit longer

-- table functions

#ipt                                              -- 0 (nothing stored)
```

```
ipt:counts()                                    -- 0 0 (ipv4_count ipv6_count)
iptable.error = nil                             -- last error message seen
for k,v in pairs(ipt) do ... end                -- iterate across k,v-pairs
for k,v in ipt:more(prefix [,true]) ... end     -- iterate across more specifics
for k,v in ipt:less(prefix [,true]) ... end     -- iterate across less specifics
for k,v in ipt:masks(af) ... end                -- iterate across masks used in af
for k,g in ipt:supernets(af) ... end            -- iterate supernets & constituents
for rdx in ipt:radixes(af [,true]) ... end      -- iterate the radix nodes
```

Notes:

- `more/less` exclude `prefix` from search results, unless 2nd arg is true
- `radixes` excludes mask nodes from iteration, unless 2nd arg is true
- `incr/decr`'s offset parameter is optional and defaults to 1
- functions return all nils on errors plus an error message
- if iterators won't iterate, check `iptable.error` for an error message
- iptable never clears the iptable.error itself

Documentation

See also the `doc` directory on github.

module constants

```
iptable.AF_INET     2
iptable.AF_INET6    10
```

module functions

Requiring `iptable` yields an object with module level functions.

```
iptable = require "iptable"
```

Prefix strings are converted to binary format following these rules:

0. A mask, if present, must be valid
1. It's treated as ipv6 if it has an ':', ipv4 otherwise
2. String/binary conversion is done by `inet_pton` & `inet_ntop`

The use of `inet_pton` implies no hexadecimal, octal or shorthand notations for ipv4.

`iptable.address(prefix)`

Returns the host address, mask length and address family for `prefix`. If `prefix` has no masklength, `mlen` will be `-1` to indicate the absence.

```
#!/usr/bin/env lua
iptable = require"iptable"
```

```
pfx4 = "10.10.10.0/19"
pfx6 = "2001:0db8:85a3:0000:0000:8a2e:0370:700/120"

print("--", iptable.address(pfx4))
print("--", iptable.address("10.10.10.10"))
print("--", iptable.address(pfx6))
print("--", iptable.address("acdc:1976::"))
print("\n-- with no spaces allowed")
print("--", iptable.address("10.10.10.10 /32"))
print("--", iptable.address(" : : /128"))
print(string.rep("-", 35))

---------- PRODUCES --------------

--  10.10.10.0  19  2
--  10.10.10.10 -1  2
--  2001:db8:85a3::8a2e:370:700 120 10
--  acdc:1976:: -1  10

-- with no spaces allowed
--  nil nil nil invalid prefix string
--  nil nil nil invalid prefix string
-----------------------------------
```

iptable.broadcast(prefix)

Applies the inverse mask to the address and returns the broadcast address, mask length and address family for `prefix`. If `prefix` has no masklength, `mlen` will be `-1` to indicate the absence and the broadcast address is the host address itself.

```
#!/usr/bin/env lua
iptable = require"iptable"
pfx4 = "10.10.10.0/19"
pfx6 = "2001:0db8:85a3:0000:0000:8a2e:0370:700/120"

print("--", iptable.broadcast(pfx4))
print("--", iptable.broadcast("10.10.10.10"))
print("--", iptable.broadcast(pfx6))
print("--", iptable.broadcast("2001:0db8::"))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  10.10.31.255    19  2
--  10.10.10.10 -1  2
--  2001:db8:85a3::8a2e:370:7ff 120 10
```

```
--  2001:db8::  -1  10
-----------------------------------
```

iptable.decr(prefix [, offset])

Decrement the ip address of the prefix (no mask is applied) and return the new ip address, mask length and address family. `offset` is optional and defaults to 1. Decrementing beyond valid address space yields `nil`.

```
#!/usr/bin/env lua
iptable = require"iptable"
pfx4 = "10.10.10.0/19"
pfx6 = "2001:0db8:85a3:0000:0000:8a2e:0370:700/120"

print("--", iptable.decr(pfx4, 1))
print("--", iptable.decr("0.0.0.0"))
print("--", iptable.decr(pfx6, 1))
print("--", iptable.decr("::"))

print(string.rep("-", 35))

---------- PRODUCES --------------
```

iptable.dnsptr(prefix [,skip])

Return a reverse dns name for the address part of the prefix, along with the prefix length seen and the address family. If `skip` evaluates to true and the prefix has a prefix length > 0, skip as many leading labels as the mask allows. On error, returns nil and an error message.

```
#!/usr/bin/env lua
iptable = require"iptable"

pfx4 = "10.10.10.0/16"
pfx6 = "2001:0db8:85a3:0000:0000:8a2e:0370:700/116"

print("--", iptable.dnsptr(pfx4))
print("--", iptable.dnsptr(pfx4, true))
print("--", iptable.dnsptr(pfx6))
print("--", iptable.dnsptr(pfx6, true))
print("--", iptable.dnsptr("2001:abcd::/32", true))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  0.10.10.10.in-addr.arpa.    16  2
--  10.10.in-addr.arpa. 16  2
```

```
--  0.0.7.0.0.7.3.0.e.2.a.8.0.0.0.0.0.0.0.0.3.a.5.8.8.b.d.0.1.0.0.2.ip6.arpa.   116 10
--  0.0.7.3.0.e.2.a.8.0.0.0.0.0.0.0.3.a.5.8.8.b.d.0.1.0.0.2.ip6.arpa. 116 10
--  d.c.b.a.1.0.0.2.ip6.arpa.   32  10
-------------------------------------
```

iptable.hosts(prefix)

Iterate across the hosts in a given prefix. Optionally include the network and broadcast addresses as well.

```
#!/usr/bin/env lua
iptable = require"iptable"

print("-- Hosts in 10.10.10.0/30:")
for pfx in iptable.hosts("10.10.10.0/30") do
    print("--", pfx)
end

print("\n-- Including netw/bcast")
for pfx in iptable.hosts("10.10.10.0/30", true) do
    print("--", pfx)
end

print(string.rep("-", 35))

---------- PRODUCES --------------

-- Hosts in 10.10.10.0/30:
--  10.10.10.1
--  10.10.10.2

-- Including netw/bcast
--  10.10.10.0
--  10.10.10.1
--  10.10.10.2
--  10.10.10.3
-----------------------------------
```

iptable.interval(start, stop)

Iterate across the subnets that cover, exactly, the ip address space bounded by the start and stop addresses.

```
#!/usr/bin/env lua
iptable = require"iptable"

for pfx in iptable.interval("10.10.10.0", "10.10.10.12") do
    print("--", pfx)
```

```
end

print(string.rep("-", 35))

---------- PRODUCES --------------

--  10.10.10.0/29
--  10.10.10.8/30
--  10.10.10.12/32
-----------------------------------
```

iptable.invert(prefix)

Invert the address of given `prefix` and return the inverted address, mask length and address family. Note: the mask is NOT applied. If that's required, convert the prefix first using `iptable.network(prefix)`.

```
#!/usr/bin/env lua
iptable = require"iptable"

print("--", iptable.invert("255.255.0.0/16"))
print("--", iptable.invert("ffff:ffff::/32"))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  0.0.255.255 16  2
--  ::ffff:ffff:ffff:ffff:ffff:ffff 32  10
-----------------------------------
```

iptable.longhand(prefix)

Explode a prefix, i.e. produce an full address string without any shorthand. Only has effect on ipv6. Embedded ipv4's are converted to hex digits as well. Any prefix length, if present, is not applied before exploding the address part of the prefix. Returns full address, prefix length and AF.

```
#!/usr/bin/env lua
iptable = require"iptable"

print("--", iptable.longhand("10.0.0.0/8"))
print("--", iptable.longhand("::"))
print("--", iptable.longhand("::ffff:11.12.13.14/128"))

print(string.rep("-", 35))

---------- PRODUCES --------------
```

```
--  10.0.0.0    8   2
--  0000:0000:0000:0000:0000:0000:0000:0000 -1  10
--  0000:0000:0000:0000:0000:ffff:0b0c:0d0e 128 10
-------------------------------------
```

iptable.mask(af, mlen [, invert])

Create a mask for the given address family **af** and specified mask length **mlen**. Use the optional 3rd argument to request an inverted mask by supplying a true value.

```
#!/usr/bin/env lua
iptable = require"iptable"

print("--", iptable.mask(iptable.AF_INET, 19))
print("--", iptable.mask(iptable.AF_INET, 19, true))
print("--", iptable.mask(iptable.AF_INET6, 91))
print("--", iptable.mask(iptable.AF_INET6, 91, true))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  255.255.224.0
--  0.0.31.255
--  ffff:ffff:ffff:ffff:ffff:ffe0::
--  ::1f:ffff:ffff
-------------------------------------
```

iptable.masklen(binary_key)

Given a binary key, **masklen** will return the number of consecutive 1-bits starting from the left. Only useful if the binary key was derived from an actual mask and not a subnet prefix. Note: the first byte of the binary is the LEN-byte of the byte array, the real key at offset 1.

```
#!/usr/bin/env lua
iptable = require"iptable"
pfx4 = "255.255.253.0"
pfx6 = "ffff:fffe::"

bin2str = function(buf)
  local s = ""
  local len = buf:byte(1)
  for i = 1, len, 1 do
    s = string.format("%s:%02x", s, buf:byte(i));
  end
  return s:sub(2) -- skip leading ':'
```

```
end

bin4, mlen4, af4 = iptable.tobin(pfx4)
bin6, mlen6, af6 = iptable.tobin(pfx6)

print("--", iptable.masklen(bin4), "consecutive 1's in:", bin2str(bin4))
print("--", iptable.masklen(bin6), "consecutive 1's in:", bin2str(bin6))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  22  consecutive 1's in: 05:ff:ff:fd:00
--  31  consecutive 1's in: 11:ff:ff:ff:fe:00:00:00:00:00:00:00:00:00:00:00
-----------------------------------
```

iptable.neighbor(prefix)

Get the adjacent subnet that, together with `prefix`, occupies their immediate parental supernet whose prefix length is 1 bit shorter. Returns the adjacent prefix, mask length and address family. Note that a prefix with a length of zero has no parental supernet.

```
#!/usr/bin/env lua
iptable = require"iptable"
pfx6 = "2001:0db8:85a3:0000:0000:8a2e:0370:700/120"

print("--", iptable.neighbor("10.10.0.0/19"))
print("--", iptable.neighbor("10.10.32.0/19"))
print("--", iptable.neighbor("10.10.10.255"))
print("--", iptable.neighbor("0.0.0.0/0"))      -- nothing larger than this
print("--", iptable.neighbor(pfx6))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  10.10.32.0  19  2
--  10.10.0.0   19  2
--  10.10.10.254    -1  2
--  nil nil nil none
--  2001:db8:85a3::8a2e:370:600 120 10
-----------------------------------
```

iptable.network(prefix)

Applies the mask to the address and returns the network address, mask length and address family for `prefix`. If `prefix` has no mask length, `mlen` will be `-1`

to indicate the absence and the network address is the host address itself.

```
#!/usr/bin/env lua
iptable = require"iptable"
pfx4 = "10.10.10.10/19"
pfx6 = "2001:0db8:85a3:0000:0000:8a2e:0370:777/120"

print("--", iptable.network(pfx4))
print("--", iptable.network("10.10.10.10"))
print("--", iptable.network(pfx6))
print("--", iptable.network("2001:0db8::"))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  10.10.0.0   19  2
--  10.10.10.10 -1  2
--  2001:db8:85a3::8a2e:370:700 120 10
--  2001:db8::  -1  10
-----------------------------------
```

`iptable.new()`

Constructor method that returns a new ipv4,ipv6 lookup table. Use it as a regular table with modified indexing:

- *exact* indexing is used for assignments or when the index has a masklength
- *longest prefix match* if indexed with a bare host address

`iptable.offset(prefix [,offset])`

Returns a new ip `address`, `masklen` and `af_family` by adding an offset to the address part of given `prefix`. The optional `offset` defaults to `1`. Returns three `nil`'s and an `error msg` on errors, such as trying to offset beyond the address family's valid address space (i.e. it won't wrap around).

```
#!/usr/bin/env lua
iptable = require"iptable"

print("--", iptable.offset("10.10.10.0/16", 10))
print("--", iptable.offset("10.10.10.0/16", -10))
print("--", iptable.offset("10.10.255.255/16"))   -- default offset 1
print("--", iptable.offset("255.255.255.255", 1)) -- yields error
print("--", iptable.offset("acdc::1976", -1))

print(string.rep("-", 35))
```

```
---------- PRODUCES --------------

--  10.10.10.10 16  2
--  10.10.9.246 16  2
--  10.11.0.0   16  2
--  nil nil nil binary operation failed
--  acdc::1975  -1  10
---------------------------------
```

iptable.properties(prefix)

Create a table with some of the properties for a given prefix. These are generally properties that require calculations, not mere lookups (multicast being an exception to this rule).

```
#!/usr/bin/env lua
iptable = require"iptable"

t, err = iptable.properties("192.168.1.1/24")
print("-- 192.168.1.1/24:")
for k,v in pairs(t) do print("   --", k, v) end
print()

t, err = iptable.properties("224.0.0.2")
print("-- 224.0.0.2:")
for k,v in pairs(t) do print("   --", k, v) end
print()

t, err = iptable.properties("::ffff:192.168.1.1/120")
print("-- ::ffff:192.168.1.1/120:")
for k,v in pairs(t) do print("   --", k, v) end
print()
t, err = iptable.properties("2001:0:4036:e378:8000:62fb:3fff:fdd2")
print("--", "2001:0:4036:e378:8000:62fb:3fff:fdd2")
for k,v in pairs(t) do print("   --", k, v) end
print(string.rep("-", 35))

---------- PRODUCES --------------

-- 192.168.1.1/24:
   --   imask   0.0.0.255
   --   class   C
   --   af  2
   --   v4mapped    ::ffff:192.168.1.1
   --   pfxlen  24
   --   v4compat    ::192.168.1.1
```

```
   --   address 192.168.1.1
   --   mask    255.255.255.0
   --   ip6to4  2002:c0a8:101::

-- 224.0.0.2:
   --   imask   0.0.0.0
   --   class   D
   --   af  2
   --   v4mapped    ::ffff:224.0.0.2
   --   pfxlen  -1
   --   v4compat    ::224.0.0.2
   --   address 224.0.0.2
   --   mask    255.255.255.255
   --   ip6to4  2002:e000:2::
   --   multicast   allrouters

-- ::ffff:192.168.1.1/120:
   --   imask   ::ff
   --   pfxlen  120
   --   address ::ffff:192.168.1.1
   --   mask    ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00
   --   af  10
   --   v4mapped    192.168.1.1

--  2001:0:4036:e378:8000:62fb:3fff:fdd2
   --   imask   ::
   --   toredo_udp  40196
   --   toredo_client   192.0.2.45
   --   af  10
   --   pfxlen  -1
   --   toredo_server   64.54.227.120
   --   address 2001:0:4036:e378:8000:62fb:3fff:fdd2
   --   mask    ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
   --   toredo_flags    32768
-----------------------------------
```

iptable.reverse(prefix)

Reverse the address byte of given `prefix` and return reversed address, mask length and address family. For ipv6, the nibbles are reversed as well. Note: any mask is NOT applied before reversal is done. If that's required, convert the prefix first using `iptable.network(prefix)`.

```
#!/usr/bin/env lua
iptable = require"iptable"
```

```
print("--", iptable.reverse("255.255.0.0"))
print("--", iptable.reverse("1112:1314:1516:1718:1920:2122:2324:2526"))

-- Note: ipv6 with 4 bytes -> reversal formatted as ipv4 in ipv6
print("--", iptable.reverse("acdc:1976::/32"))


print(string.rep("-", 35))

---------- PRODUCES --------------

--  0.0.255.255 -1  2
--  6252:4232:2212:291:8171:6151:4131:2111  -1  10
--  ::103.145.205.202   32  10
-----------------------------------
```

iptable.size(prefix)

Return the number of hosts covered by given `prefix`. Since ipv6 subnets might have more than 2^52 hosts in it, this function uses Lua arithmatic to yield the number.

```
#!/usr/bin/env lua
iptable = require"iptable"
pfx6 = "2001:0db8:85a3:0000:0000:8a2e:0370:700/120"

print("--", iptable.size("10.10.10.0/24"))
print("--", iptable.size("10.10.0.0/31"))
print("--", iptable.size(pfx6))
print("--", iptable.size("2001::/0"))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  256.0
--  2.0
--  256.0
--  3.4028236692094e+38
-----------------------------------
```

iptable.split(prefix)

Split a prefix into its two subnets. Returns both network addresses of the two subnets, along with the new prefix length and AF. In case of errors, such as trying to split a host address, it returns all nils and an error message.

```lua
#!/usr/bin/env lua
iptable = require"iptable"

print("--", iptable.split("10.0.0.0/8"))
print("--", iptable.split("10.10.10.10/32"))
print("--", iptable.split("acdc:1979::/32"))
print("--", iptable.split("acdc:1979::"))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  10.0.0.0    10.128.0.0  9   2
--  nil nil nil nil prefix already at max length
--  acdc:1979:: acdc:1979:8000::    33  10
--  nil nil nil nil prefix already at max length
-----------------------------------
```

iptable.subnets(prefix [, mlen])

Iterate across the subnets in a given prefix. The optional new mask length defaults to being 1 longer than the mask in given prefix. Returns each subnet as a prefix. In case of errors, iptable.error provides some information.

```lua
#!/usr/bin/env lua
iptable = require"iptable"

prefix = "10.10.10.0/28"
print("-- /30 subnets in " .. prefix)
for pfx in iptable.subnets(prefix, 30) do
    print("-- +", pfx)
end

prefix = "acdc:1976::/30"
print("-- /32 subnets in " .. prefix)
for pfx in iptable.subnets(prefix, 32) do
    print("-- +",  pfx)
end

print(string.rep("-", 35))

---------- PRODUCES --------------

-- /30 subnets in 10.10.10.0/28
-- +    10.10.10.0/30
-- +    10.10.10.4/30
-- +    10.10.10.8/30
```

```
-- +    10.10.10.12/30
-- /32 subnets in acdc:1976::/30
-- +    acdc:1974::/32
-- +    acdc:1975::/32
-- +    acdc:1976::/32
-- +    acdc:1977::/32
-----------------------------------
```

iptable.tobin(prefix)

Returns the binary key used internally by the radix tree for a string key like `prefix`. It's a length encoded byte string, i.e. the first byte represents the length of the entire byte string and the remaining bytes are from the prefix itself. Useful if the convenience functions fall short of what needs to be done, or to figure out the mask length of a regular mask.

```
#!/usr/bin/env lua
iptable = require"iptable"
pfx4 = "10.10.0.0/19"
pfx6 = "2001:0db8:85a3:0000:0000:8a2e:0370:700/120"

bin2str = function(buf)
  local s = ""
  local len = buf:byte(1)
  for i = 1, len, 1 do
    s = string.format("%s:%02x", s, buf:byte(i));
  end
  return s:sub(2) -- skip leading ':'
end

bin4, mlen, af = iptable.tobin(pfx4)
bin6, mlen, af = iptable.tobin(pfx6)

print("--", bin2str(bin4))
print("--", bin2str(bin6))
print("--", iptable.masklen(iptable.tobin("255.255.255.252")))

print(string.rep("-", 35))

---------- PRODUCES --------------

--  05:0a:0a:00:00
--  11:20:01:0d:b8:85:a3:00:00:00:00:8a:2e:03:70:07:00
--  30
-----------------------------------
```

iptable.toredo(...)

Create a table with toredo details from either a single ipv6 address or a list of arguments that include the server, client, udp port and flags.

```
#!/usr/bin/env lua
iptable = require"iptable"

t, err = iptable.toredo("64.54.227.120", "192.0.2.45", 40196, 2^15)
for k,v in pairs(t) do print("--   compose", k, v) end
print()

t, err = iptable.toredo(t.ipv6)
for k,v in pairs(t) do print("-- decompose", k, v) end

print(string.rep("-", 35))

---------- PRODUCES --------------

--   compose    udp 40196
--   compose    server  64.54.227.120
--   compose    flags   32768
--   compose    client  192.0.2.45
--   compose    ipv6    2001:0:4036:e378:8000:62fb:3fff:fdd2

-- decompose    udp 40196
-- decompose    server  64.54.227.120
-- decompose    flags   32768
-- decompose    client  192.0.2.45
-- decompose    ipv6    2001:0:4036:e378:8000:62fb:3fff:fdd2
-----------------------------------
```

iptable.tostr(binary_key)

The reciprocal to `tobin` turns a binary key back into a string key. Note that binary keys cannot be used to index into an iptable instance.

```
#!/usr/bin/env lua
iptable = require"iptable"
pfx4 = "255.255.253.0"
pfx6 = "ffff:fffe::"

bin2str = function(buf)
  local s = ""
  local len = buf:byte(1)
  for i = 1, len, 1 do
    s = string.format("%s:%02x", s, buf:byte(i));
  end
```

```
  return s:sub(2) -- skip leading ':'
end

bin4, mlen4, af4 = iptable.tobin(pfx4)
bin6, mlen6, af6 = iptable.tobin(pfx6)

print("--", iptable.tostr(bin4), " == ", pfx4)
print("--", iptable.tostr(bin6), " == ", pfx6)

print(string.rep("-", 35))

---------- PRODUCES --------------

--  255.255.253.0    ==     255.255.253.0
--  ffff:fffe::  ==     ffff:fffe::
-----------------------------------
```

table functions

Basic operations

An iptable behaves much like a regular table, except that it ignores non-ipv4 and non-ipv6 keys silently, when assigning to a key it is always interpreted as a subnet (missing masks are added as max masks for the address family in question), lookups are exact if the key has a mask and only use longest prefix match when the lookup key has no mask. For assignments, the mask (as supplied of as a default value) is always applied before storing the key, value pair in the internal radix tree(s). Hence, iterating across an iptable always shows keys to be actual subnets with a mask, in CIDR notation.

```
#!/usr/bin/env lua
acl = require"iptable".new()
acl["10.10.10.0/24"] = true   -- match the /24
acl["10.10.10.8/30"] = false  -- except this /30

print("-- 1 exact match for prefix 10.10.10.0/24  -", acl["10.10.10.0/24"])
print("-- 2 longest prefix match for 10.10.10.9   -", acl["10.10.10.9"])
print("-- 3 longest prefix match for 10.10.10.100 -", acl["10.10.10.100"])
print("-- 4 exact match for prefix 10.10.10.10/30 -", acl["10.10.10.10/30"])
print("-- 5 acl number of entries:", #acl)

print(string.rep("-", 35))

---------- PRODUCES --------------

-- 1 exact match for prefix 10.10.10.0/24  -    true
-- 2 longest prefix match for 10.10.10.9   -    false
-- 3 longest prefix match for 10.10.10.100 -    true
```

```
-- 4 exact match for prefix 10.10.10.10/30 -    false
-- 5 acl number of entries: 2
-----------------------------------
```

ipt:more(prefix [,inclusive])

Given a certain `prefix`, which need not be present in the iptable, iterate across more specific subnets actually present in the table. The optional second argument will include the given search `prefix` if found in the results, if its value is true. The default value for `inclusive` is `false`.

```
#!/usr/bin/env lua
ipt = require"iptable".new()
ipt["10.10.0.0/16"] = 1
ipt["10.10.9.0/24"] = 2
ipt["10.10.10.0/24"] = 3
ipt["10.10.10.0/25"] = 4
ipt["10.10.10.0/26"] = 5
ipt["10.10.10.128/30"] = 6

-- search more specifics
for pfx in ipt:more("10.10.10.0/24") do
    print("-- exclusive search", pfx)
end
print()

-- includes search prefix (if present)
for pfx in ipt:more("10.10.10.0/24", true) do
    print("-- inclusive search", pfx)
end

print(string.rep("-", 35))

---------- PRODUCES --------------

-- exclusive search 10.10.10.0/26
-- exclusive search 10.10.10.0/25
-- exclusive search 10.10.10.128/30

-- inclusive search 10.10.10.0/26
-- inclusive search 10.10.10.0/25
-- inclusive search 10.10.10.0/24
-- inclusive search 10.10.10.128/30
-----------------------------------
```

ipt:less(prefix)

Given a certain `prefix`, which need not be present in the iptable, iterate across less specific subnets actually present in the table. The optional second argument will include the given search `prefix` if found in the results, if its value is true. The default for `inclusive` is `false`.

```
#!/usr/bin/env lua
ipt = require"iptable".new()

ipt["10.10.0.0/16"] = 1
ipt["10.10.9.0/24"] = 2
ipt["10.10.10.0/24"] = 3
ipt["10.10.10.0/25"] = 4
ipt["10.10.10.0/26"] = 5
ipt["10.10.10.128/30"] = 6

-- search less specifics only
for pfx in ipt:less("10.10.10.0/25") do
    print("-- exclusive search", pfx)
end
print()

-- include starting search prefix (if present)
for pfx in ipt:less("10.10.10.0/25", true) do
    print("-- inclusive search", pfx)
end

print(string.rep("-", 35))

---------- PRODUCES --------------

-- exclusive search 10.10.10.0/24
-- exclusive search 10.10.0.0/16

-- inclusive search 10.10.10.0/25
-- inclusive search 10.10.10.0/24
-- inclusive search 10.10.0.0/16
-----------------------------------
```

ipt:supernets(af)

Iterate across pairs of subnets present in the iptable that could be combined into their parent supernet. The iterator returns the supernet in CIDR notation and a regular table that contains the key,value pairs for both the supernet's constituents as well as the supernet itself, should that exist (which need not be the case). Useful when trying to minify a list of prefixes.

```lua
#!/usr/bin/env lua
iptable = require "iptable"
ipt = iptable.new()

ipt["10.11.0.0/16"] = 1
ipt["10.10.9.0/24"] = 2
ipt["10.10.10.0/24"] = 3
ipt["10.10.10.0/25"] = 4
ipt["10.10.10.128/25"] = 5
ipt["10.10.10.0/30"] = 6
ipt["10.10.10.4/30"] = 7

-- find adjacent prefixes
for supernet, grp in ipt:supernets(iptable.AF_INET) do
    print("-- supernet", supernet)
    for subnet, val in pairs(grp) do
        print("   --", subnet, val)
    end
end

print(string.rep("-", 35))

---------- PRODUCES --------------

-- supernet 10.10.10.0/29
   --   10.10.10.4/30   7
   --   10.10.10.0/30   6
-- supernet 10.10.10.0/24
   --   10.10.10.128/25 5
   --   10.10.10.0/25   4
   --   10.10.10.0/24   3
-- supernet 10.10.10.0/29
   --   10.10.10.0/30   6
   --   10.10.10.4/30   7
-- supernet 10.10.10.0/24
   --   10.10.10.128/25 5
   --   10.10.10.0/25   4
   --   10.10.10.0/24   3
-----------------------------------
```

`ipt:masks(af)`

An `iptable` utilizes two radix trees internally, one for ipv4 subnets and one for ipv6 subnets. `ipt:masks(af)` will iterate across the actual masks used in the tree for the given address family `af`. No idea when this might be useful, but there you have it.

```lua
#!/usr/bin/env lua
iptable = require "iptable"
ipt = iptable.new()

ipt["10.10.0.0/16"] = 1
ipt["10.10.9.0/24"] = 2
ipt["10.10.10.0/24"] = 3
ipt["10.10.10.0/25"] = 4
ipt["10.10.10.0/26"] = 5
ipt["10.10.10.128"] = 6   -- same as "10.10.10.128/32"

-- iterate across masks in the trie
for mask in ipt:masks(iptable.AF_INET) do
    print("--", mask)
end

print(string.rep("-", 35))

---------- PRODUCES --------------

--  255.255.0.0
--  255.255.255.0
--  255.255.255.128
--  255.255.255.192
--  255.255.255.255
-----------------------------------
```

ipt:counts()

Returns the number of ipv4 subnets and ipv6 subnets present in the iptable. Internally, the number of prefixes per tree is kept in a `size_t` counter which may overflow if you go crazy on the number of ipv6 prefixes.

```lua
#!/usr/bin/env lua
iptable = require "iptable"
ipt = iptable.new()

ipt["10.10.0.0/16"] = 1
ipt["10.10.9.0/24"] = 2
ipt["10.10.10.0/24"] = 3
ipt["10.10.10.0/25"] = 4
ipt["10.10.10.0/26"] = 5
ipt["10.10.10.128/30"] = 6
ipt["2001::dead:beef/120"] = 1
ipt["2002::dead:beef/120"] = 2
ipt["2003::dead:beef/120"] = 3
```

```lua
af4, af6 = ipt:counts()

print("--", af4, "ipv4 prefixes")
print("--", af6, "ipv6 prefixes")

print(string.rep("-", 35))

---------- PRODUCES --------------

--  6   ipv4 prefixes
--  3   ipv6 prefixes
-----------------------------------
```

ipt:radixes(af[, masktree])

Iterate across the radix nodes of the radix tree for the given address family `af`. Only really useful to graph the radix trees while debugging or for educational purposes. The radix nodes are returned by the iterator encoded as Lua tables. More information in the ***Lua stack functions*** section in `lua_iptable.c.md` (or its pdf) in the doc-folder. For example code, check `ipt2dot.lua` on github in the `src/lua` folder.

```lua
#!/usr/bin/env lua
iptable = require "iptable"
ipt = iptable.new()

ipt["10.10.0.0/16"] = 1
ipt["2003::dead:beef/120"] = 1

-- dump radix nodes in the radix trie (key-tree only)
for rdx in ipt:radixes(iptable.AF_INET | iptable.AF_INET6) do
    print("-- tree exclusive", rdx._NAME_)
end
print()

-- also include the mask-tree
for rdx in ipt:radixes(iptable.AF_INET | iptable.AF_INET6, true) do
    print("-- tree inclusive", rdx._NAME_)
end

print(string.rep("-", 35))

---------- PRODUCES --------------

-- tree exclusive   RADIX_NODE_HEAD
-- tree exclusive   RADIX_NODE
-- tree exclusive   RADIX_NODE
```

```
-- tree inclusive   RADIX_NODE_HEAD
-- tree inclusive   RADIX_NODE
-- tree inclusive   RADIX_NODE
-- tree inclusive   RADIX_MASK_HEAD
-- tree inclusive   RADIX_NODE
-- tree inclusive   RADIX_NODE
-----------------------------------
```

Radix tree graphs

iptable's github repo has two additional, small lua modules that can be used to dump a radix tree to a dot-file for conversion by graphviz:

- `ipt2dot`, dumps the tree with full radix node details
- `ipt2smalldot`, dumps only the key-tree with less node details

An iptable has two radix trees, one for ipv4 and one for ipv6. Each radix tree actually consists of both a radix tree for the keys being stored as well as a radix tree for the masks being used in the tree (to save memory consumption). `ipt2dot` graphs, for a given AF, the key-tree by default and includes the mask-tree only when its optional 3rd argument is true. `ipt2smalldot` however, only graphs the key-tree and shows less detail of the radix nodes in the tree.

For large trees, this gets pretty messy but for small trees the images are legible. Primarily for fun with no real application other than a means to assist during development.

IPv4 tree

```
#!/usr/bin/env lua
iptable = require "iptable"
dotify = require "src.lua.ipt2dot"

ipt = iptable.new()
ipt["10.10.10.0/24"] = 1
ipt["10.10.10.0/25"] = 2
ipt["10.10.10.128/26"] = 3
ipt["11.11.11.0/24"] = 4

imgfile = arg[1]
dotfile = imgfile:gsub("...$", "dot")
dottext = dotify(ipt, iptable.AF_INET, true)

fh = io.open(dotfile, "w")
fh:write(table.concat(dottext, "\n"))
fh:close()
```

```
os.execute(string.format("dot -Tpng %s -o %s", dotfile, imgfile))
print(string.rep("-",35))

---------- PRODUCES --------------
```

IPv6 tree

```
#!/usr/bin/env lua
iptable = require "iptable"
dotify = require "src.lua.ipt2dot"

ipt = iptable.new()
ipt["acdc:1974::/32"] = "Can I sit next to you?"
ipt["acdc:1976::/32"] = "Jailbreak"
ipt["acdc:1979::/32"] = "Highway to hell"
ipt["acdc:1980::/32"] = "Touch too much"

imgfile = arg[1]
dotfile = imgfile:gsub("...$", "dot")
dottext = dotify(ipt, iptable.AF_INET6, true)

fh = io.open(dotfile, "w")
fh:write(table.concat(dottext, "\n"))
fh:close()

os.execute(string.format("dot -Tpng %s -o %s", dotfile, imgfile))
print(string.rep("-",35))

---------- PRODUCES --------------
```

Alternate IPv4 tree

```
#!/usr/bin/env lua
iptable = require "iptable"
dotify = require "src.lua.ipt2smalldot"

ipt = iptable.new()
ipt["10.10.10.0/24"] = 1
ipt["10.10.10.0/25"] = 2
ipt["10.10.10.128/25"] = 3
ipt["10.10.10.128/26"] = 4
ipt["11.11.11.0/24"] = 2

imgfile = arg[1]
dotfile = imgfile:gsub("...$", "dot")
```

```
dottext = dotify(ipt, iptable.AF_INET)

fh = io.open(dotfile, "w")
fh:write(table.concat(dottext, "\n"))
fh:close()

os.execute(string.format("dot -Tpng %s -o %s", dotfile, imgfile))
print(string.rep("-",35))

---------- PRODUCES --------------
```

Alternate IPv6 tree

```
#!/usr/bin/env lua
iptable = require "iptable"
dotify = require "src.lua.ipt2smalldot"

ipt = iptable.new()
ipt["acdc:1974::/32"] = "Can I sit next to you?"
ipt["acdc:1976::/32"] = "Jailbreak"
ipt["acdc:1979::/32"] = "Highway to hell"
ipt["acdc:1980::/32"] = "Touch too much"

imgfile = arg[1]
dotfile = imgfile:gsub("...$", "dot")
dottext = dotify(ipt, iptable.AF_INET6)

fh = io.open(dotfile, "w")
fh:write(table.concat(dottext, "\n"))
fh:close()

os.execute(string.format("dot -Tpng %s -o %s", dotfile, imgfile))
print(string.rep("-",35))

---------- PRODUCES --------------
```

Example code

Minify list of prefixes

First keep merging subnets into their parental supernet, until no merging takes place anymore. Then remove any remaining subnets that weren't merged but lie inside another subnet in the table.

```
#!/usr/bin/env lua
iptable = require "iptable"
```

```lua
-- fill a table with list of prefixes
ipt = iptable.new()
acl = {
    "10.10.10.0/30", "10.10.10.0/25", "10.10.10.128/26", "10.10.10.192/26",
    "10.10.11.0/25", "10.10.11.128/25",
    "11.11.11.0", "11.11.11.1", "11.11.11.2", "11.11.11.3", "11.11.11.4"
}

-- load up the table
for _, pfx in ipairs(acl) do ipt[pfx] = true end
for k,_ in pairs(ipt) do print("-- original", k) end
print()

-- subnets unite!
changed = true
while (changed) do
    changed = false
    for supernet, grp in ipt:supernets(iptable.AF_INET) do
        for subnet, _ in pairs(grp) do ipt[subnet] = nil end
        ipt[supernet] = true -- it may have been included in grp
        changed = true
    end
end

-- don't sweat the small stuff
for prefix, _ in pairs(ipt) do
    for subnet, _ in ipt:more(prefix) do ipt[subnet] = nil end
end

for k, _ in pairs(ipt) do print("-- minified", k) end
print(string.rep("-",35))

---------- PRODUCES --------------

-- original 10.10.10.0/30
-- original 10.10.10.0/25
-- original 10.10.10.128/26
-- original 10.10.10.192/26
-- original 10.10.11.0/25
-- original 10.10.11.128/25
-- original 11.11.11.0/32
-- original 11.11.11.1/32
-- original 11.11.11.2/32
-- original 11.11.11.3/32
-- original 11.11.11.4/32
```

```
-- minified 10.10.10.0/23
-- minified 11.11.11.0/30
-- minified 11.11.11.4/32
----------------------------------
```