In [1]:

```
# Set up environment with correct dependencies
using Pkg
Pkg.activate(".")
Pkg.instantiate()
```

```
 Activating environment at `~/GitHub/MathSys/teaching/MA934/MA934-slid
es/Project.toml`
```

In [2]:

```
using Plots
using LaTeXStrings
using Revise
pyplot()
# Set default fonts for all plots
fnt = Plots.font("DejaVu Sans", 8.0)
default(titlefont=fnt, guidefont=fnt, tickfont=fnt, legendfont=fnt)
```

MA934

Data types and data structures

There is more to life than linear arrays...

Data types

A *data type* is an attribute of data that tells the compiler/interpreter how that data will be used. For example, `Float64` and `Int64` are both 64-bit binary strings but are interpreted differently.

Primitive types: `Int64` etc, `Float32` etc, `Bool` , `Char`

Composite types: derived from multiple primitive types: `Array` , `struct` .

Julia also provides some special types: `Any` , `Nothing` , `Union` - see the documentation (https://docs.julialang.org/en/v1/base/base/#Special-Types) for others.

Working with types

Julia provides functions for type checking that can be very useful:

- `typeof(x)` : returns the type of x
- `isa(x, T)` : checks if x has type T

In [3]:

```
n = Int64(10)
x = Float64(10.0)
println((typeof(n), typeof(x)))
```

```
(Int64, Float64)
```

In [4]:

```
println((isa(x, Int64), isa(x,Float64
```

```
(false, true)
```

Note `DataType` is itself a type:

In [5]:

```
typeof(typeof(x))
```

Out[5]:

```
DataType
```

The `Nothing` special type

Confusingly, `Nothing` is a type that can only take the special value `nothing`. This represents the value returned by functions which do not return anything.

In [6]:

```
nada = Nothing()
println("Type of nada: ", typeof(nada), ",  Value of nada : ", nada)
```

```
Type of nada: Nothing,  Value of nada : nothing
```

Similar to the `NULL` value in C or `None` in Python.

The `Union` special type

The `Union` is a type that includes all instances of any of its argument types.

In [7]:

```
intOrString = Union{Int64, String}
n = Int64(42)
s = "Hello world"
println((typeof(n), typeof(s)))
println((isa(n, intOrString), isa(s, intOrString)))

```

```
(Int64, String)
(true, true)
```

Particularly useful are unions like `Union(Float64, Nothing)` to represent the possibility of absent values.

Composite data types

A collection of named fields, that can be treated as a single value.

- Defined in Julia using the `struct` keyword:
- By default a struct cannot be changed once instantiated.
- Use `mutable struct` keyword to change this.

In [8]:

```julia
module tmp

mutable struct point
    x::Float64
    y::Float64
    label::String
end

end
P = tmp.point(1.0, 2.0, "point A")
println("x = ", P.x, ",  y = ", P.y,
```

```
x = 1.0,  y = 2.0, label is:
point A
```

In [9]:

```julia
P.x = -1.0
```

Out[9]:

```
-1.0
```

Constructors

- A *constructor* is a function called to initialise a struct.
- A default constructor is defined automatically:

  ```
  point(::Float64, ::Float64,
   ::String)
  ```

- Two types: *outer* and *inner*.

We can define our own outer constructors like any other function:

In [10]:

```julia
point(x::Float64) = tmp.point(x,x,"")
```

Out[10]:

```
point (generic function with
1 method)
```

In [11]:

```julia
P = point(1.0)
P
```

Out[11]:

```
Main.tmp.point(1.0, 1.0, "")
```

Inner constructors

1. are declared *inside* the type declaration.
2. have access to special function `new` that creates an uninitialised instance of the type.

Useful for enforcing constraints and building self-referential objects.

In [12]:

```julia
module tmp
struct OrderedPair
    x::Real
    y::Real
    OrderedPair(x,y) = x > y ?
        println("out of order") :
        new(x,y)
end
end
```

```
WARNING: replacing module tm
p.
```

Out[12]:

```
Main.tmp
```

In [13]:

```julia
x = tmp.OrderedPair(2.3, 3.0)
```

Out[13]:

```
Main.tmp.OrderedPair(2.3, 3.
0)
```

In [14]:

```
### More explicit version of the inner constructor example
module tmp
struct OrderedPair
    x::Real
    y::Real

    function OrderedPair(x,y)
        println("This is the inner constructor")
        if x > y
            println("out of order")
        else
            return new(x,y)
        end
    end

end
end

x = tmp.OrderedPair(1.0, 2.0)
```

```
This is the inner constructor

WARNING: replacing module tmp.
```

Out[14]:

```
Main.tmp.OrderedPair(1.0, 2.0)
```

Data structures?

A data structure is a specialised way of organising data in a computer so that certain operations can be performed efficiently.

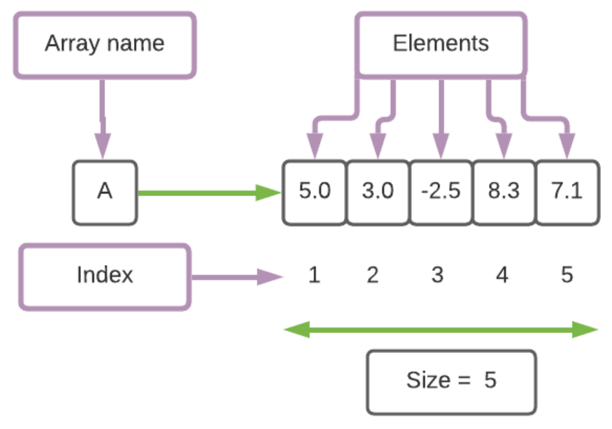
- Composite types are simplest examples.
- *Static* data structures have a fixed size. *Dynamic* data structures can grow and shrink depending on the data that they contain.
- Associated with almost every data structure is a set of basic operations that it is designed to perform efficiently (conversely some other operations might be very inefficient or impossible.)

Examples of some common data structures

- Linear arrays
- Linked lists
- Stacks
- Queues

- Hash tables
- Binary trees
- Heaps
- Graphs

Arrays



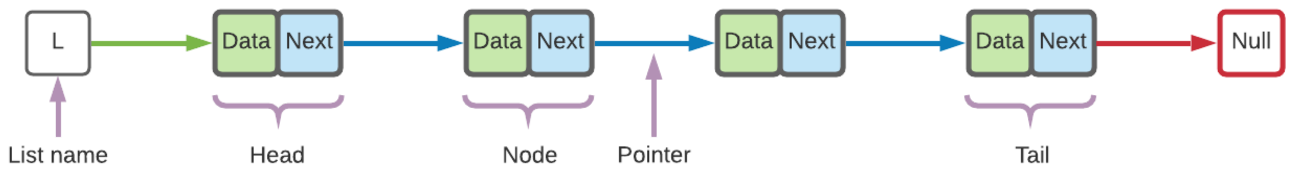
Basic operations:

- access(i) : return get value at index i
- update(i,v) : set value at index i equal to v.

insert() and delete() not possible - static data structure.

Building block for many other data structures.

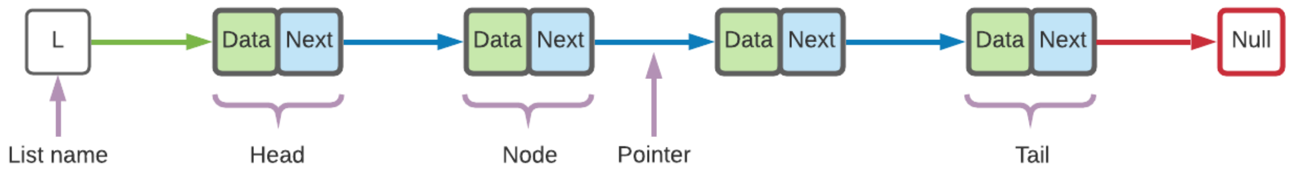
Linked lists



A linked list is a sequence of elements called *nodes* in linear order that are linked to each other.

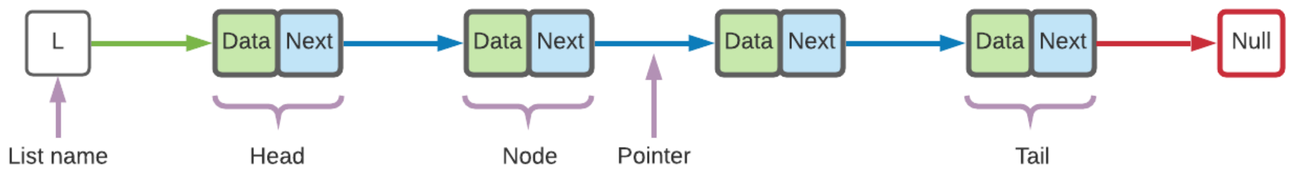
The first/last node is called the *head/tail* respectively.

Linked lists



- Each node consists of a data container and a link to the next node.
- Dynamic data structure but only sequential access is possible.
- Variants: singly linked, doubly linked, circularly linked.

Linked lists : basic operations



- search(x): determine if data x is in the list (and perhaps return a reference to it).
- insert(x): add new node with data x at beginning/middle/end.
- delete(x): delete node with data x from the list.

Aside: pointers and references

Discussions of linked lists often refer to linking nodes using *pointers*. A pointer (especially in C/C++) is a data type that contains the memory address of another object/variable.

Julia does not have pointers - variables are accessed via *references*.

A reference is also a data type that contains the memory address of another object/variable.

Aside: pointers and references - so what's the difference?

- A reference must refer to an existing object. It cannot change once created.
- A pointer can be NULL and can be updated to refer to a different memory location by changing its value.

Pointers are powerful but dangerous:

- segmentation faults
- memory leaks
- dangling pointers

If Maslov (https://en.wikipedia.org/wiki/Law_of_the_instrument) were a software engineer:

"When the only tool you have is C++, every problem looks like your thumb".

Stacks

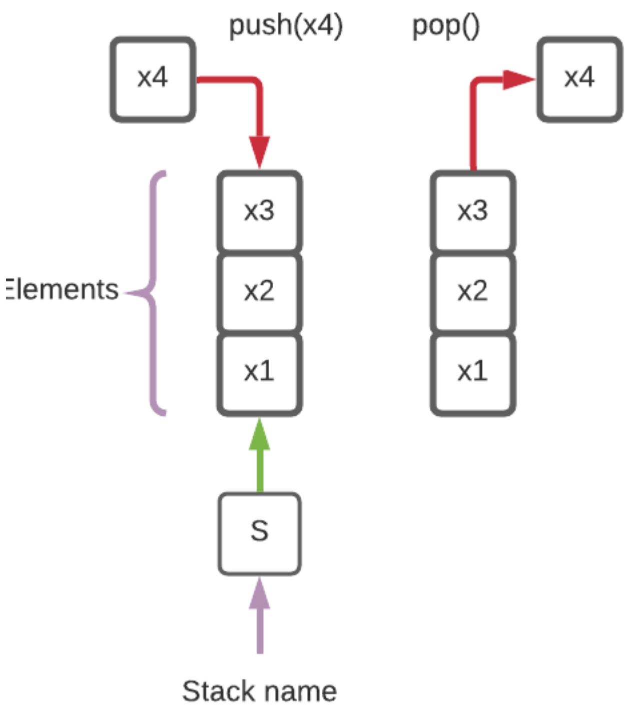
A *stack* is a linear data store with a LIFO (Last In First Out) access protocol: the last inserted element must be accessed first.

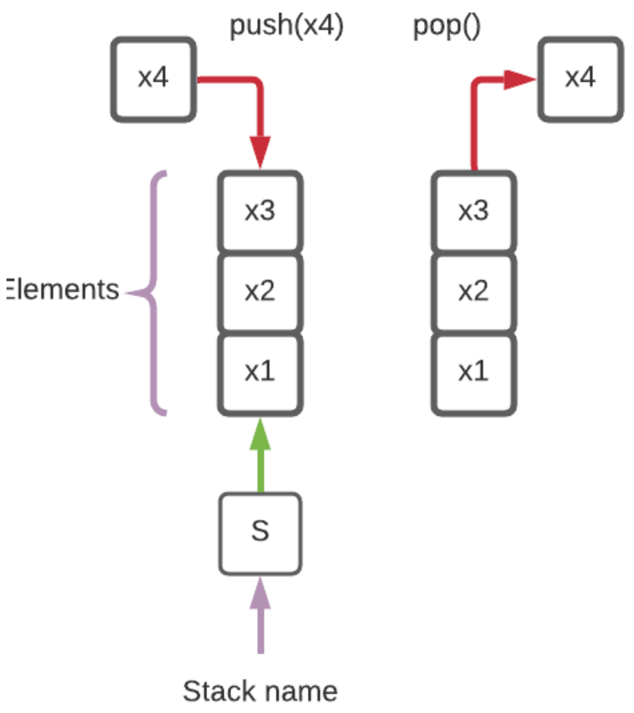
Can be static or dynamic.

So named because it resembles a stack of plates...

Used, for example, to implement function calls in recursive programming.

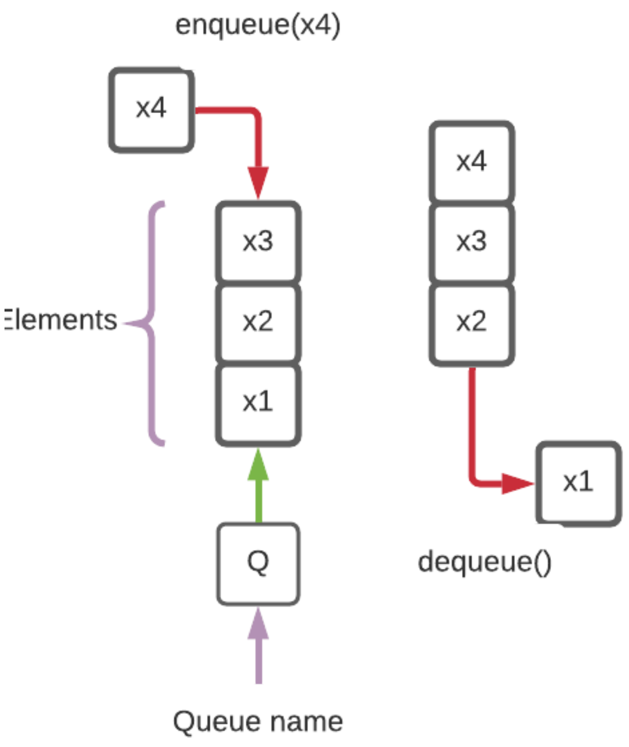
Stacks : basic operations





- push(x) : add the element x to the top of the stack.
- pop() : remove the top element from the stack and return it.
- peek() : return the top elemt from the stack without deleting it.
- isempty() : check if the stack is empty.

Queues



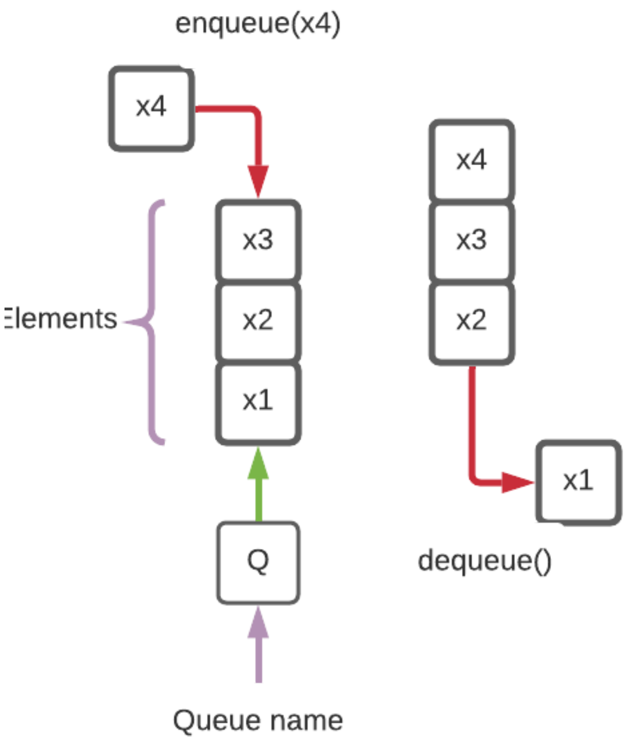
A *queue* is a linear data store with a FIFO (First In First Out) access protocol: the first inserted element must be accessed first.

Can be static or dynamic.

So named because it resembles a real queue!

Used, for example, to serve requests on a shared resource.

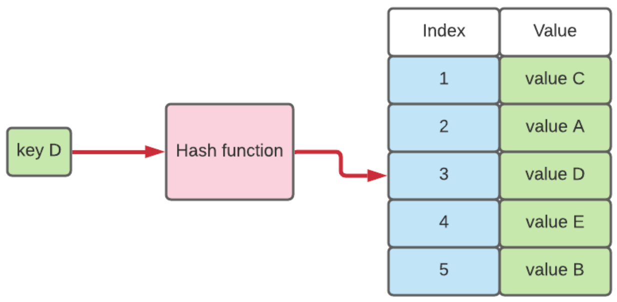
Queues : basic operations



- enqueue(x): insert element x to the end of the queue.

Hash tables (also associative array or dictionary)

- dequeue(): return the element at the beginning of the queue and delete it from the queue.



A hash table stores a set of values,

$$\{A, B, C, D, E\},$$

associated with a set of keys,

$$\{key\ A, key\ B, key\ C, key\ D, key\ E\},$$

in a way that supports efficient lookup - i.e. $\mathcal{O}(1)$.

Direct addressing (convert key X to an integer, k, and store value X in slot k) is often not feasible.

Hash tables - an example

Suppose the keys are integers in the range 1 - 1024 and we need to store, say, 4 random key-value pairs.

- Direct addressing would require an array of size 1024.
- Instead use an array of size 23 and the hash function

$$h(k) = k\%23 + 1$$

In [15]:

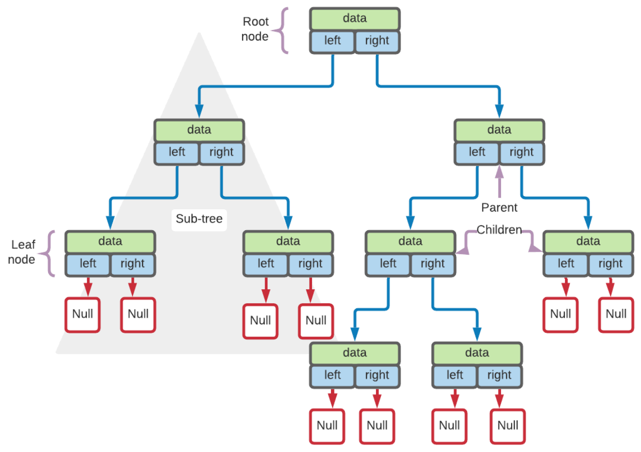
```
keys = rand(0:1024, 4)
idx = [k%23 + 1 for k in keys]
for i in 1:4
    println("Key ", keys[i], " -> ",
end
```

```
Key 127 ->  index 13
Key 342 ->  index 21
Key 43 ->  index 21
Key 553 ->  index 2
```

Of course need a strategy to resolve conflicts. e.g. use buckets.

Probability of conflicts grows as the *load factor* (# entries/#buckets) increases.

Binary trees



A binary tree is a hierarchical data structure in which nodes are linked together in parent/child relationships.

Each node contains a data container and pointers/references to left and right child nodes.

Structural recursion

Note that linked lists and binary trees have a recursive aspect:

- A linked list is either empty, or a node followed by a list.
- A binary tree is either empty, or a node with two binary trees as children.

Such data structures can actually be *defined* in a self-referential way. This is called *structural recursion*.

Structural recursion in Julia

Recursive definition of the linked list type:

```
mutable struct LinkedList
    data::Any
    next::LinkedList
end
```

In [16]:

```
module tmp
mutable struct LinkedList
    data::Any
    next::LinkedList
end
end
```

```
WARNING: replacing module tm
p.
```

Out[16]:

```
Main.tmp
```

Doesn't quite work due to initialisation problem:

In [17]:

```
L = tmp.LinkedList(0.0,nothing)
```

```
MethodError: Cannot `convert
` an object of type Nothing
to an object of type Main.tm
p.LinkedList
Closest candidates are:
  convert(::Type{T}, !Matche
d::T) where T at essentials.
jl:171
  Main.tmp.LinkedList(::Any,
!Matched::Main.tmp.LinkedLis
t) at In[16]:3
  Main.tmp.LinkedList(::Any,
!Matched::Any) at In[16]:3

Stacktrace:
 [1] Main.tmp.LinkedList(::F
loat64, ::Nothing) at ./In[1
6]:3
 [2] top-level scope at In[1
7]:1
 [3] include_string(::Functi
on, ::Module, ::String, ::St
ring) at ./loading.jl:1091
```

Structural recursion in Julia:

Method 1: use the `Union` type:

Method 2: use an inner constructor:

```
mutable struct LinkedList
    data::Any
    next::Union{LinkedList, Nothi
ng}
end
```

Worksheet 3 demonstrates this method.

```
mutable struct LinkedList
  data::Any
  next::LinkedList

  LinkedList() = (x=new();
  x.next=x; x)

  LinkedList(d::Any,L::LinkedLis
t)    = new(d,L)
end
```

Method 2 implementation - type definition

In [18]:

```
mutable struct LinkedList
    data::Any
    next::LinkedList
    LinkedList() = (x=new(); x.next=x; return x)
    LinkedList(d::Any, L::LinkedList) = new(d,L)
end


```

Method 2 implementation - function to add item

In [19]:

```
function add!(L::LinkedList, d::Any)
    global L = LinkedList(d, L)
end
```

Out[19]:

```
add! (generic function with 1 method)
```

Method 2 implementation - function to remove item

In [20]:

```
function pop!(L::LinkedList)
    if L.next == L
        println("List is empty")
        return
    else
        d = L.data
        global L = L.next
        return d
    end
end
```

Out[20]:

```
pop! (generic function with 1 method)
```

Method 2 implementation - function to print all items

In [21]:

```
function Base.print(L::LinkedList)
    if L.next == L
        return
    else
        print(L.next)
        println(L.data)
    end
end
```

Testing the code

In [22]:

```
data = ['J','u','l','i','a']
L = LinkedList()
L2 = LinkedList("Data", LinkedList())
L2
```

Out[22]:

```
LinkedList("Data", LinkedList(#undef, LinkedList(#= circular reference
@-1 =#)))
```

In [23]:

```
for item in data
    println(item)
    add!(L,item)
end
L
```

```
J
u
l
i
a
```

Out[23]:

```
LinkedList('a', LinkedList('i', LinkedList('l', LinkedList('u', Linked
List('J', LinkedList(#undef, LinkedList(#= circular reference @-1 =
#)))))))
```

In [24]:

```
print(L)
```

```
J
u
l
i
a
```

Testing the code

In [25]:

```
pop!(L)
print(L)
```

```
J
u
l
i
```

This is actually a list-based implementation of a stack.

Binary search trees (BST)

A BST stores integer keys in a sorted order to facilitate fast search:

- All nodes, y, in left subtree of any node, x, have y.key ≤ x.key.
- All nodes, y, in the right subtree of any node x, have y.key ≥ x.key.

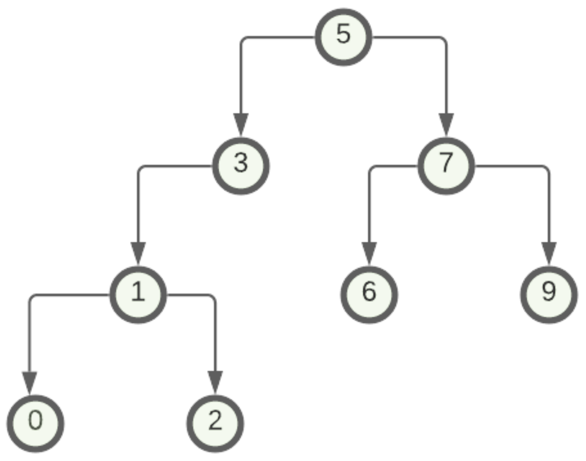
Here is a BST storing the keys {0,1,2,3,5,6,7,9}

Binary search trees (BST)

A BST stores integer keys in a sorted order to facilitate fast search:

- Nodes, y, in left subtree of node, x, have y.key ≤ x.key.
- Nodes, y, in the right subtree of node x, have y.key ≥ x.key.

Here is a another BST storing the keys {0,1,2,3,5,6,7,9}



Not unique.

Fast search :

Recursive algorithm to search for a key in a BST.

Maximum number of comparisons is the depth of the tree.

If the tree is *balanced*, depth is $\mathcal{O}(\log_2 n)$.

Note *building* the tree is $\mathcal{O}(n)$

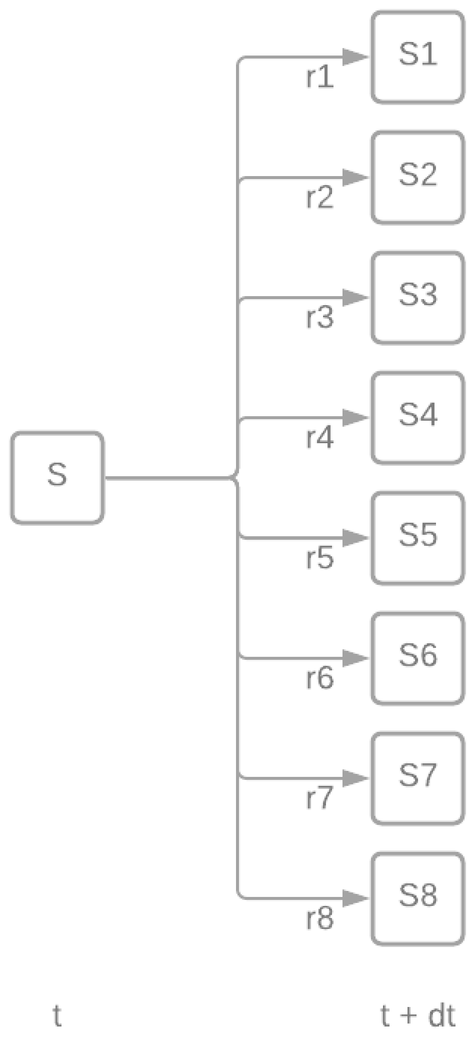
```
search(T::BST, k::int)
  if T is empty
    return false
  elseif T.key == k
    return true
  else
    if k <= T.key
      search(T.left, k)
    else
      search(T.right, k)
    end
  end
end
```

Another application: event selection in the Gillespie algorithm

Simulates trajectories from a continuous time Markov chain.

From S at time t, 8 possible states, $S_1 \dots S_8$, accessible with transition rates, $r_1 \dots r_8$.

Probability of transition $S \to S_i$ is proportional to r_i.

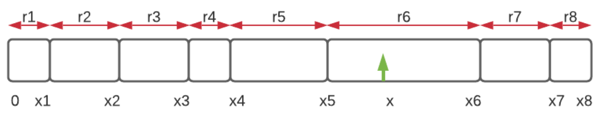


Gillespie algorithm

Build the list of partial sums:

$$x_i = \sum_{j=1}^{i} r_j$$

Generate $x \sim \mathrm{Uniform}(0, x_8)$



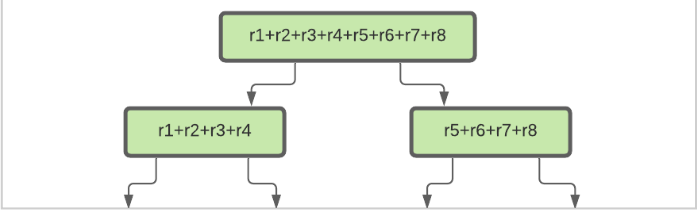
Find which interval x falls in: find k such that $x_{k-1} \leq x < x_k$.

Update state $S \to S_k$ and update time $t \to t + \Delta t$ where $\Delta t \sim \mathrm{Exponential}(x_8)$.

In practice number of transitions, n, large. Can we find k faster than $\mathcal{O}(n)$?

Interval membership problem.

Fenwick trees



Each node in a Fenwick tree stores the sum of the values stored in its children.

Leaf nodes also need to store an integer key identifying the interval.

Similar to tree search but when descending the right subtree, must remember to exclude the partial sum on the left subtree.

Fast interval membership

```
search(T::FT, x::Float)
  if T is leaf
    return T.key
  else
    if x <= T.left.value
      search(T.left, x)
    else
      search(T.right, x - T.left.
value)
    end
  end
end
```

If the tree is balanced, this search is $\mathcal{O}(\log_2 n)$ (depth of tree).

Gotcha? Transition rates usually depend on state. Reconstructing the tree at each step would be $\mathcal{O}(n)$.

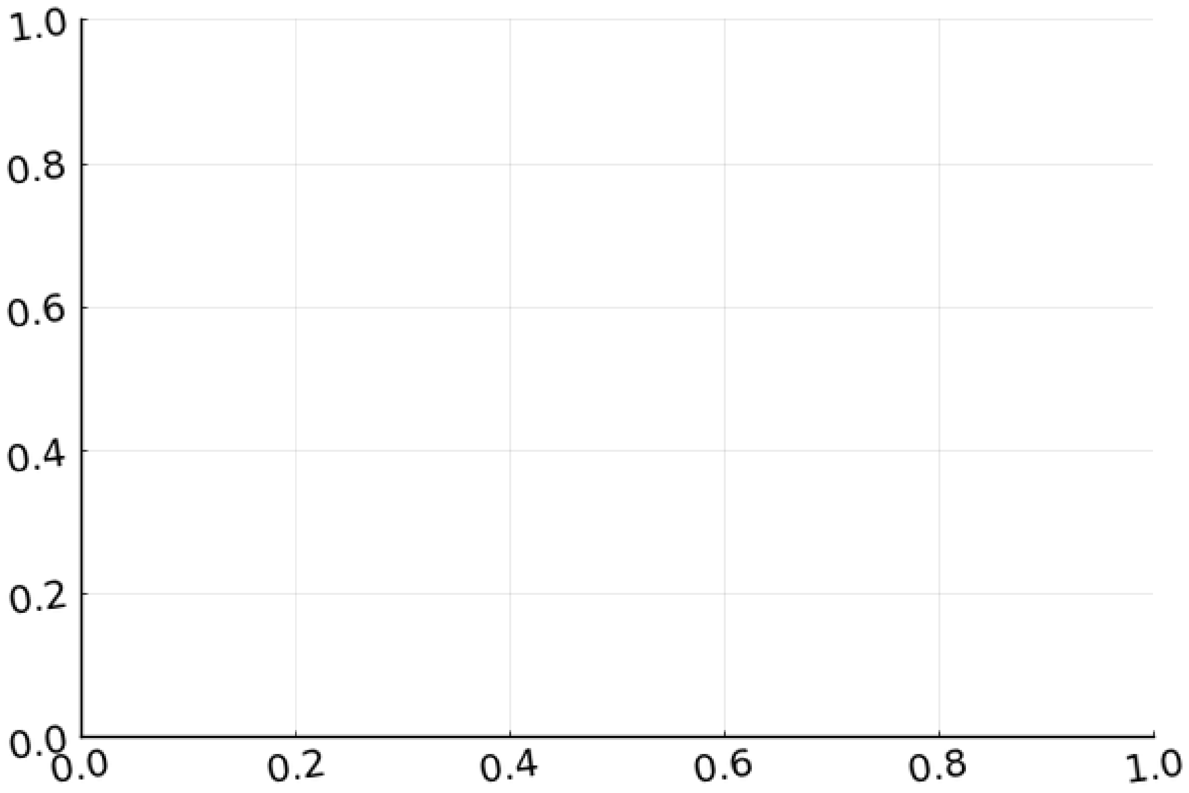
Partial sums can be updated in $\mathcal{O}(\log_2 n)$ operations. OK if small number of rates change at each step.

Need occasional rebalancing.

In [26]:

```
plot()
```

Out[26]:



In [ ]:

```

```