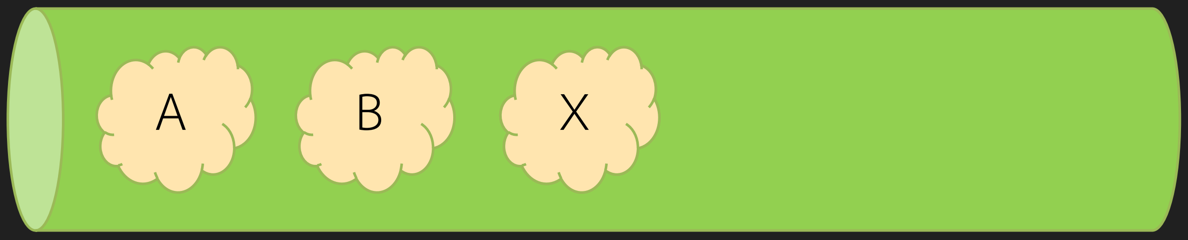
Queue

First-in-First-out data structure

Nattee Niparnan

Intro

- Just like a normal queue in real life.
  - First-in-First-out data structure
  - One way in (back of queue), one way out (front of queue)
  - push = add data to the back of the queue
  - pop = remove data from the head of the queue



Front Back

push("A")

push("B")

push("C")

pop()

push("X")

pop()

pop()

Basic

```cpp
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

int main() {
    queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);
    while (q.empty() == false) {
        cout << q.front() << endl;
        q.pop();
    }
    cout << "-- example 2 --" << endl;
    queue<vector<int>> q2;
    vector<int> v1 = {1,2,3};
    vector<int> v2 = {99,88,-1};
    q2.push( v1 );
    q2.push( v2 );
    cout << q2.back()[1] << endl;
    cout << q2.front().size() << endl;
    auto x = q2.front();
    q2.pop();
    cout << x[0] << endl;

}
```

size_t	q.size()
bool	q.empty()
void	q.push(T data)
void	q.pop()
T	q.front()
T	q.back()

Limitation

- Same limitation as stack
    - No iterator
        - No begin(), end()
        - Can only access front and back of the queue
        - If we wish to access all members, we have to pop it all
    - Do not call front(), back(), pop() when the queue is empty

Radix Sort

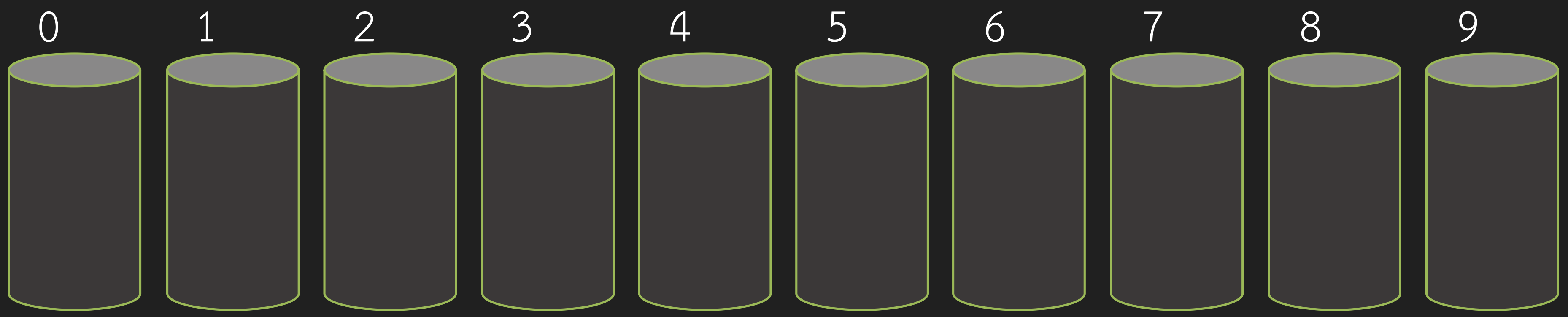
Queue Application: Fast sorting with no comparison

Overview

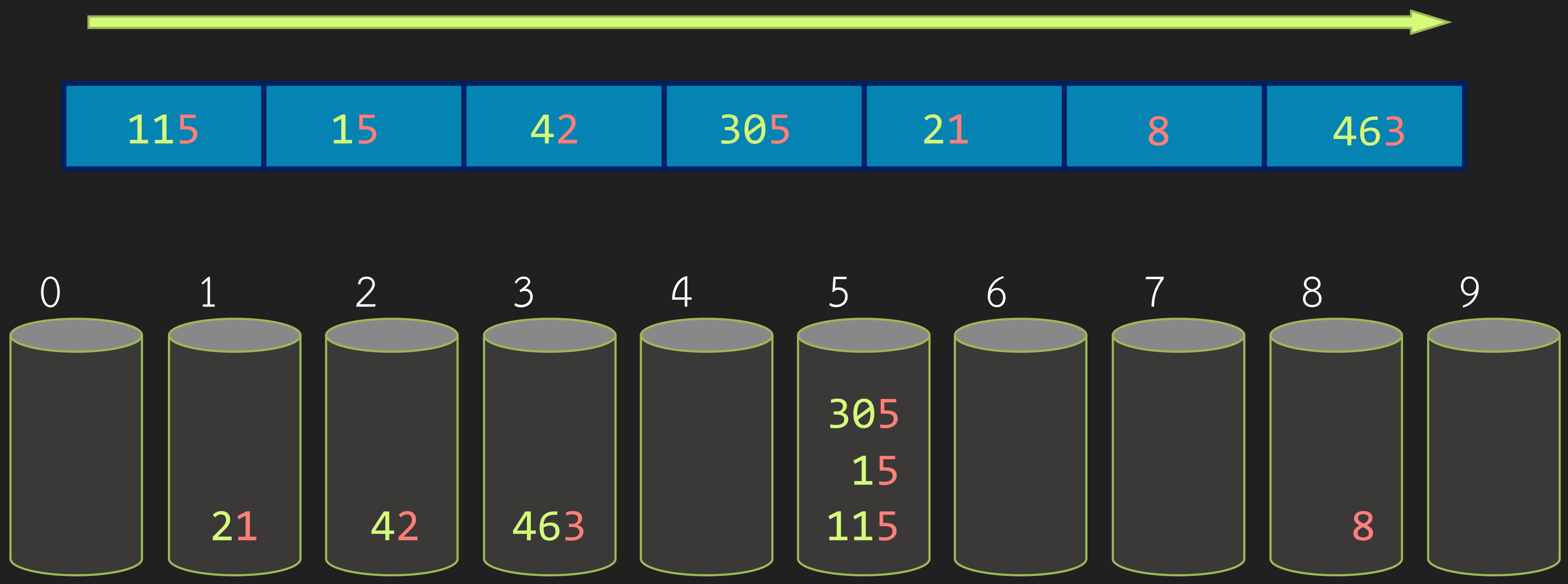
- Put all data in an array
- For each digit X, from LSD to MSD
  - PUT TO QUEUE step: Sort by digit X by putting all of data from the array into B queues
    - B is the base of the number
    - For example, for a base 10 number, we will have Queue[0] to Queue[9]
    - Put into the queue labelled with that digit
  - GET FROM QUEUE step: Start from queue 0 to queue B-1, remove data from the queue and put back to the array

Example:

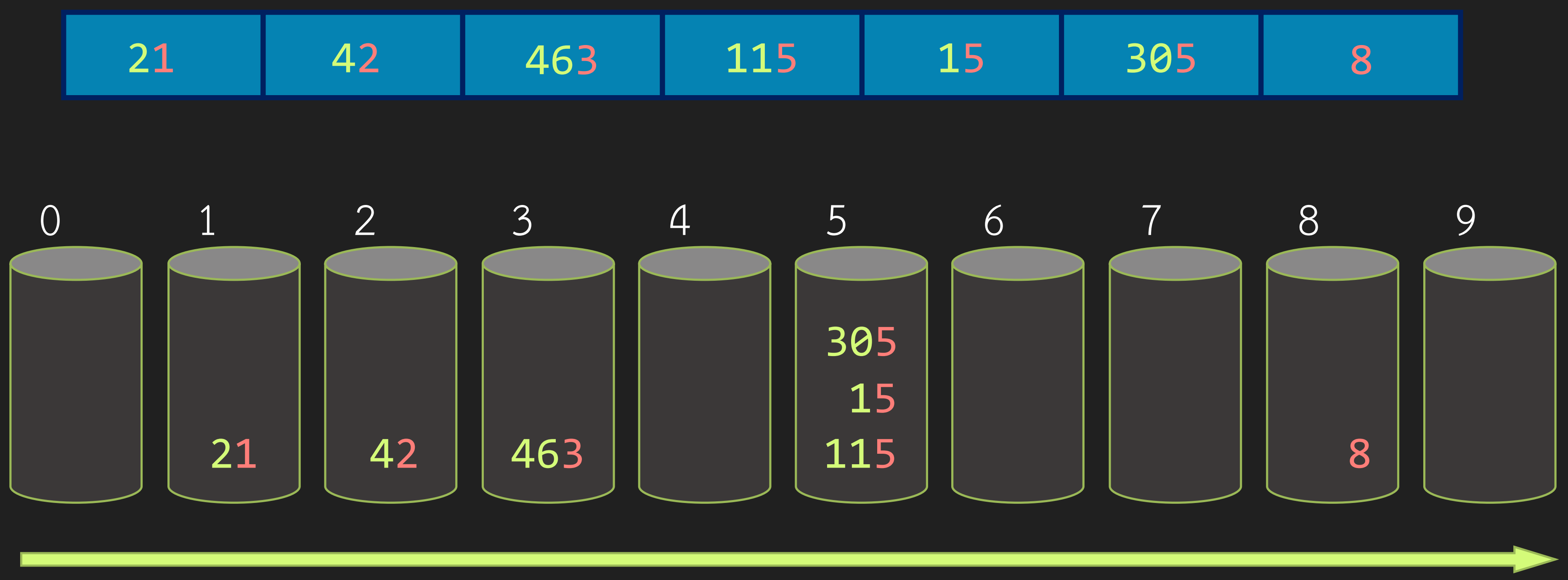
115	15	42	305	21	8	463



Example: Round 1 (digit 0), to queue



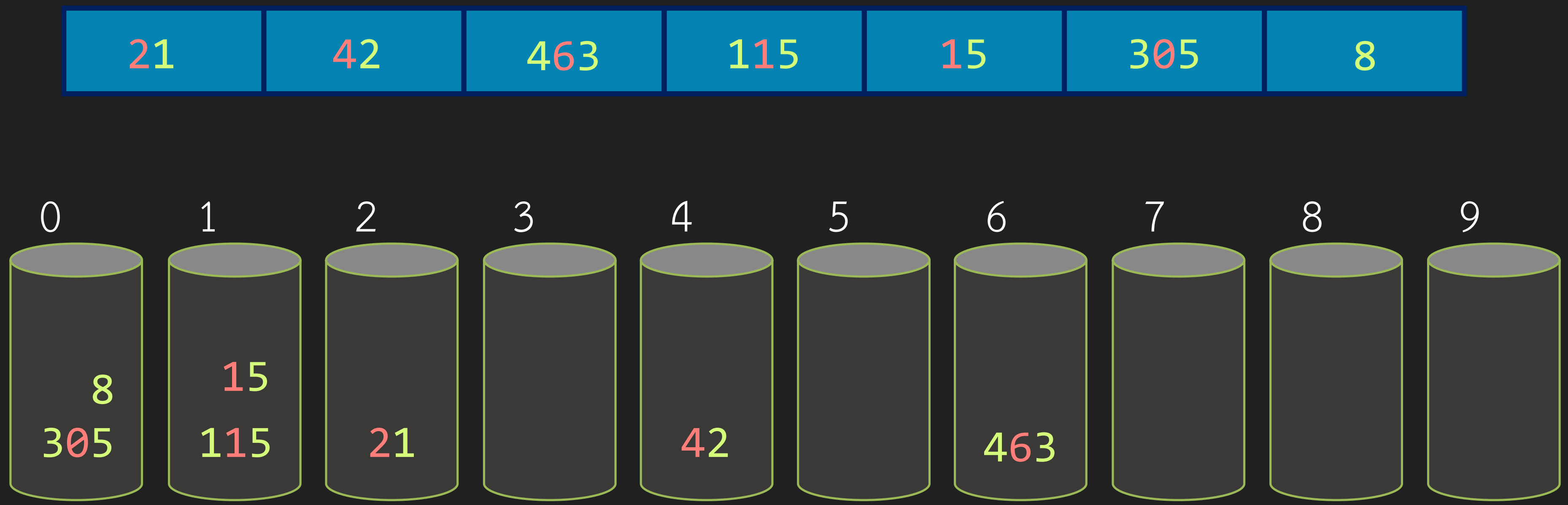
Example: Round 1 (digit 0), from queue



Data is now sorted by the last digits,

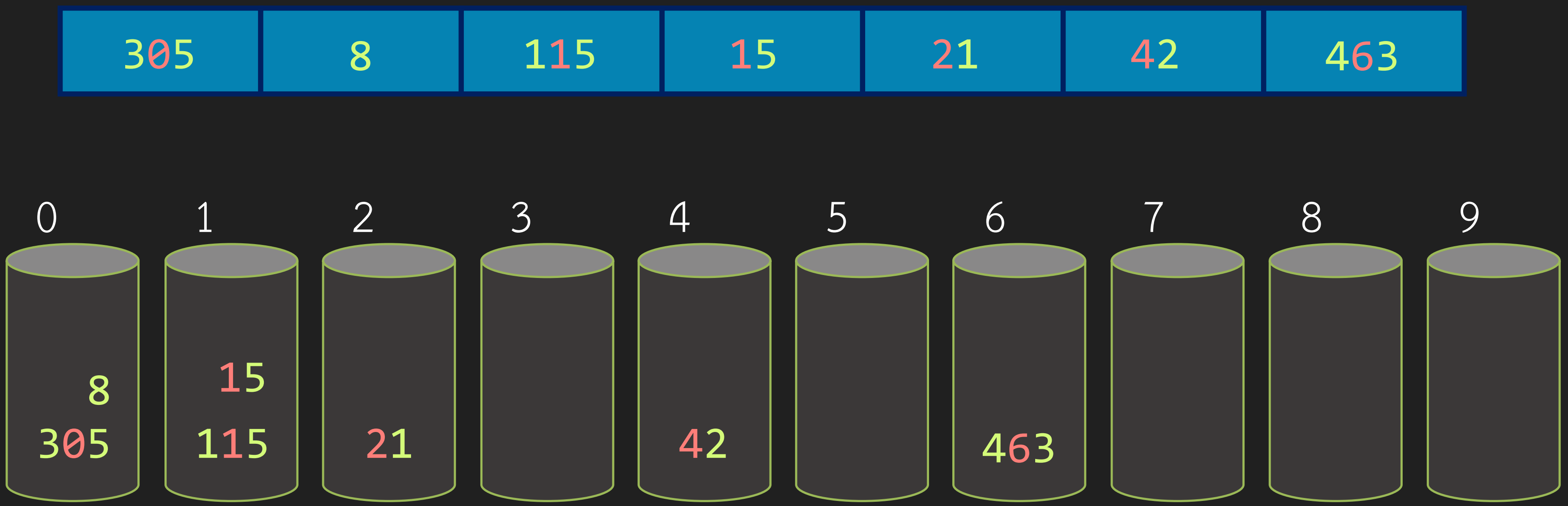
because we pop from queue 0 to 9

Example: Round 2 (digit 1), to queue



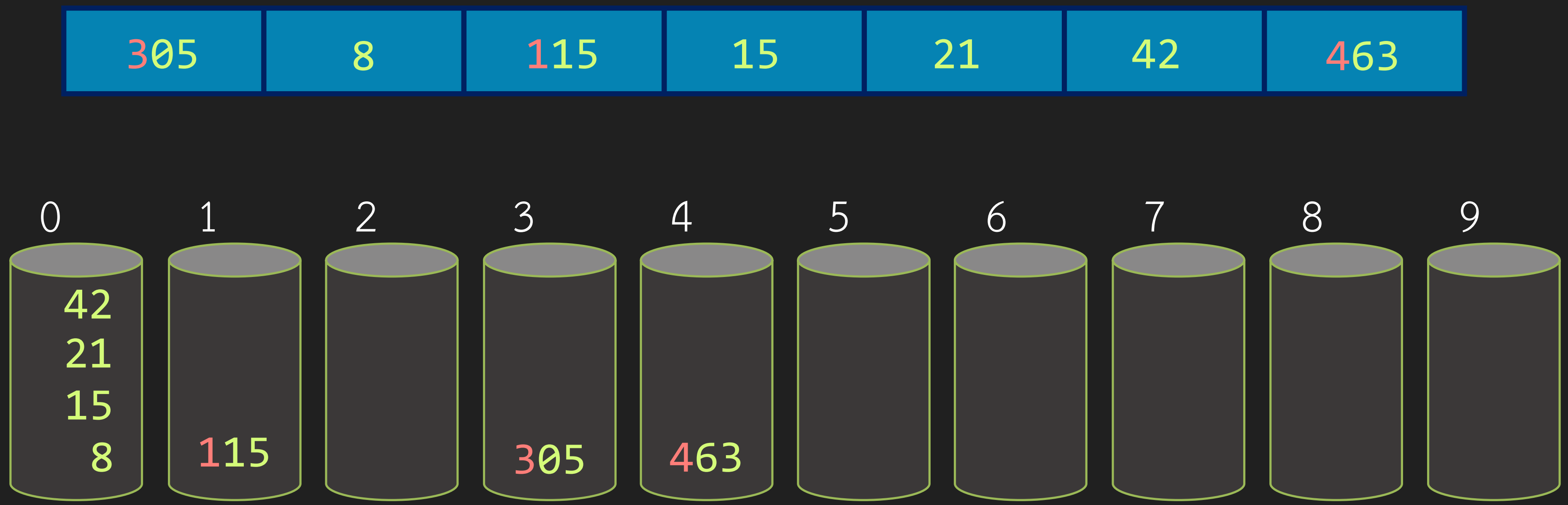
In each queue, the data is sorted by the last digit, because we goes from left to right in the array which is already sorted by the last digit

Example: Round 2 (digit 1), from queue



Data is now sorted by last two digits, because we goes from queue 0 to 9, which is grouped by digit 1 and the data in each queue is sorted by the last digit.

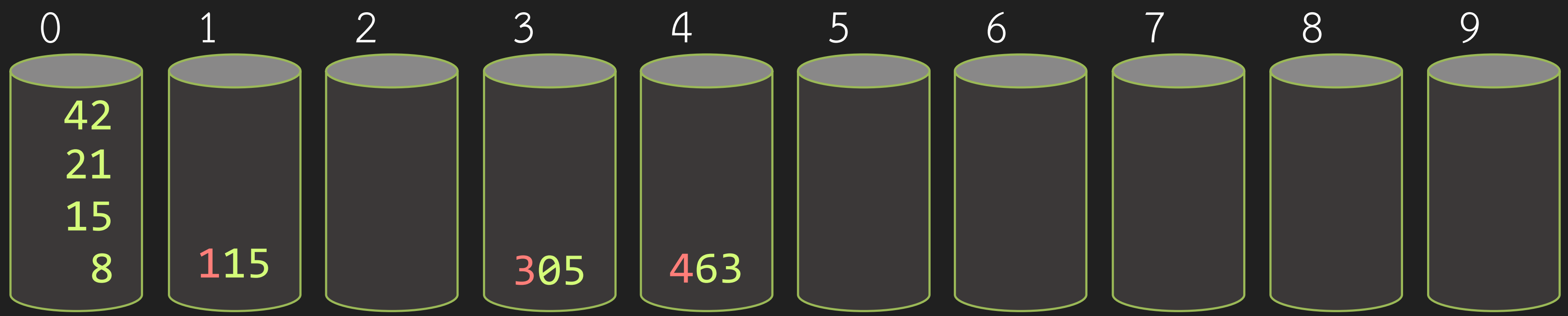
Example: Round 3 (digit 2), to queue



In each queue, the data is sorted by the last two digits, because we goes from left to right in the array which is already sorted by the last two digits

Example: Round 3 (digit 2), from queue

8	15	21	42	115	305	463



Data is now sorted by all digits, because we goes from queue 0 to 9, which is grouped by digit 2 and the data in each queue is sorted by the last two digits.

Code

```
#define base 10

int getDigit(int v, int k) {
    // return the kth digit of v (MSD is digit 0)
    int i;
    for (i=0; i<k; i++) v /= base;
    return v % base;
}

// d = number of digits
void radixSort(vector<int> &data,int d) {
    queue<int> q[base];
    for (int k=0; k<d; k++) {
        for (auto &x : data)
            q[getDigit(x,k)].push(x);
        for (int i=0,j=0; i<base; i++)
            while(!q[i].empty()) {
                data[j++] = q[i].front(); q[i].pop();
            }
    }
}
```

Breadth First Search

Queue Application: Gotta generate ’em all!

The Problem

- Given a positive integer X
- Start with a number 1, find a sequence of arithmetics operations, either "* 3", or "/ 2" that makes 1 into X
  - the / 2 is integer division, e.g., 5 / 3 = 1 (not 1.6666)
  - The sequence must be as short as possible
- Example
  - 10 = 1 * 3 * 3 * 3 * 3 / 2 / 2 / 2
  - 31 = 1 * 3 * 3 * 3 * 3 * 3 / 2 / 2 / 2 / 2 / 2 * 3 * 3 / 2

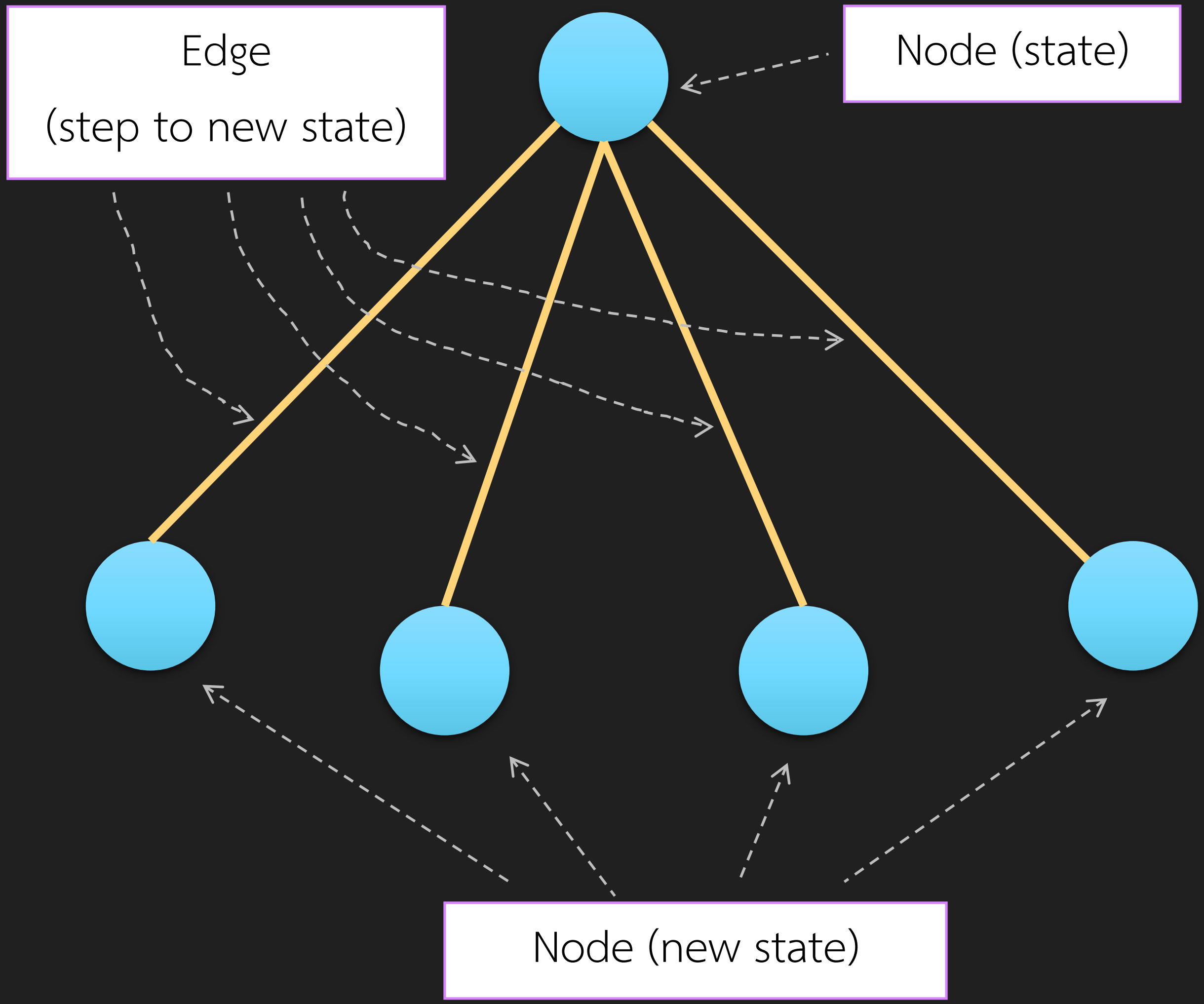
The Idea

- Generate all possible sequences
  - Start with length 1, 2, 3, … until we find one that gives X
- This is called an exhaustive search algorithm
  - Systematically enumerate all possible somethings

Tree Structure

- A structure to illustrates search algorithm
- Divide into steps
  - Start with initial solution
  - For each possible outcome (called a state) of each step, genenate all proper possible next step
    - Also, check if the current step is what we need
- Written as a diagram of node and edge

Enumerate



- Write a diagram, each state is a node (drawn as a circle)
- Enumerate all possible next steps of each state as edges (drawn as a line)
    - Doing each step will result in a new state

Example

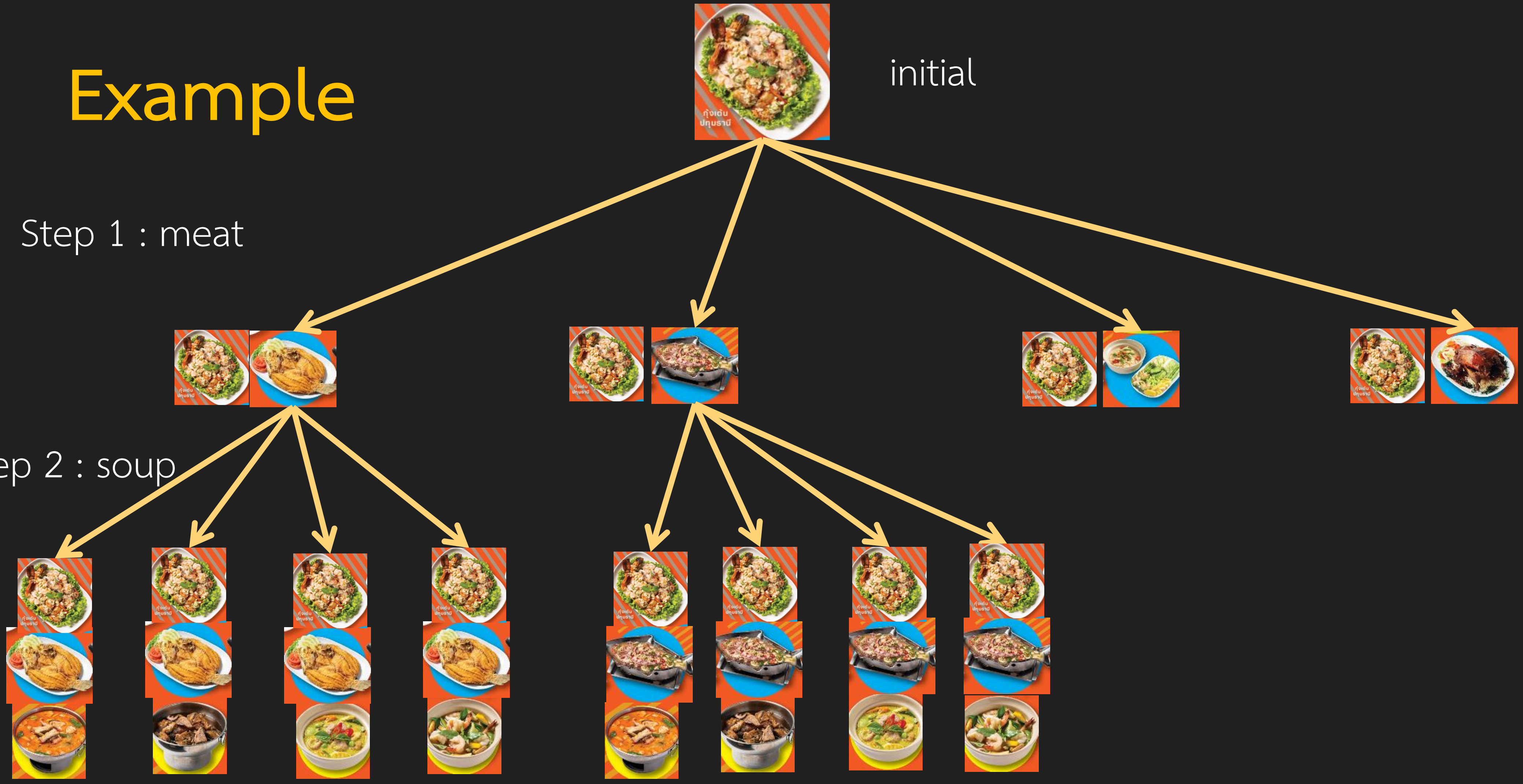
- Enumerate all possible meals from this promotion
  - There are 3 steps: Meat, Soup, Veggie
  - Each step we have to pick one
- Initial solution = starting order
- Each step, pick one of the choice and put into order

Example

initial

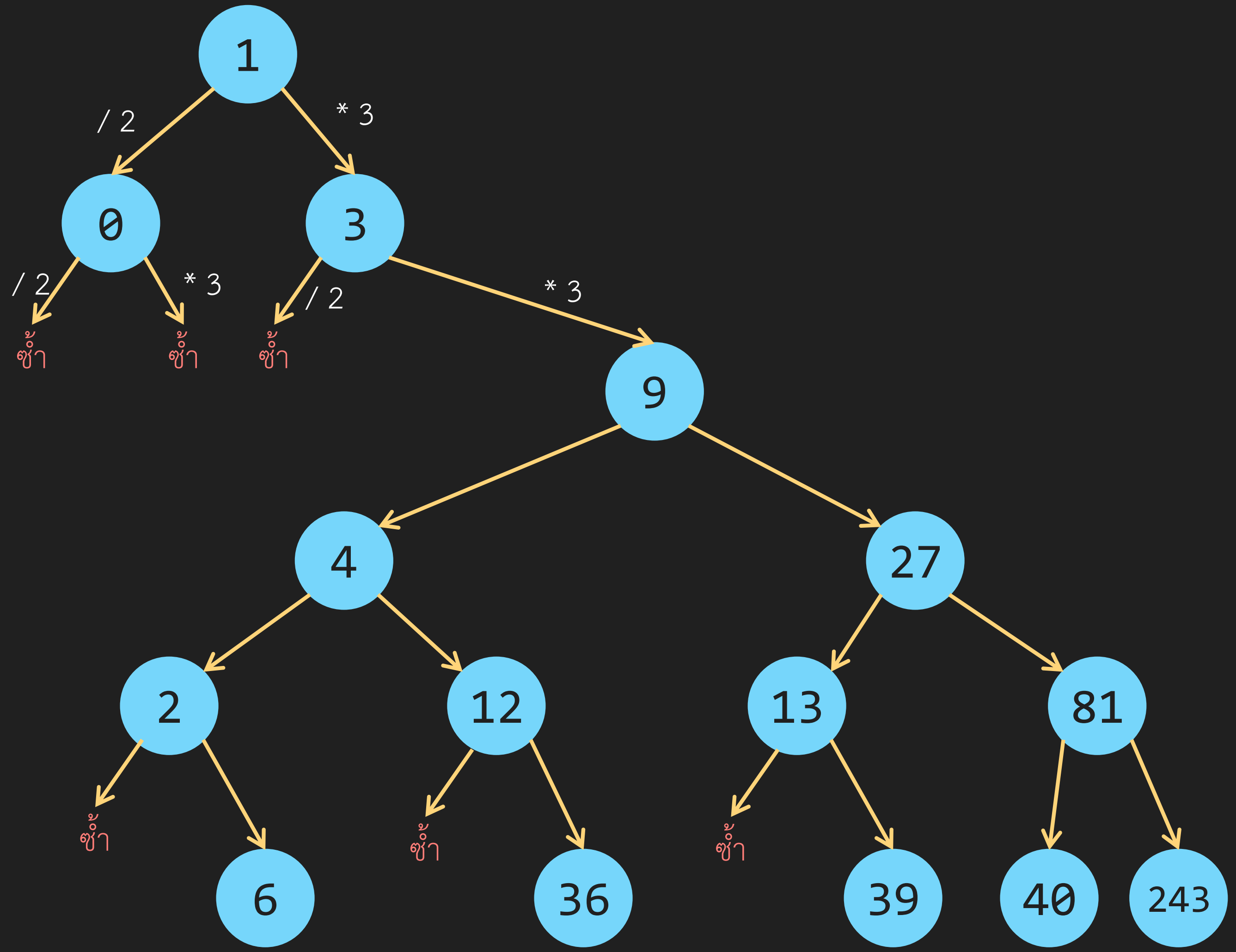
Step 1 : meat

Step 2 : soup



Back to our problem

- Start with 1
- Each step is either * 3 or / 2
- Issue: might get repeated number
    - Solution: if we have found it, do not generate new step



Code

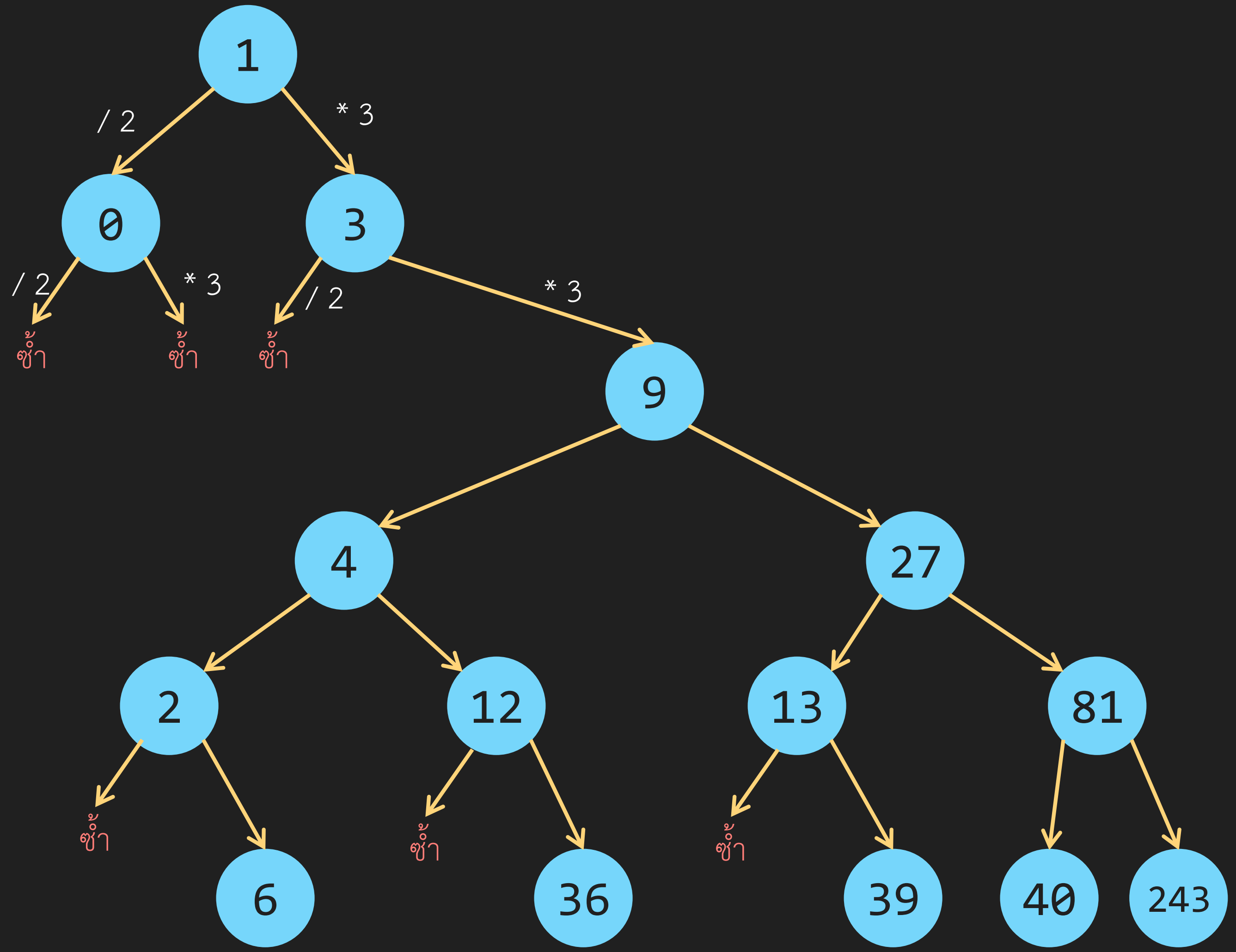
```
void m3d2(int target) {
    map<int, int> prev;
    queue<int> q;
    int v = 1;
    q.push(1); prev[1] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        if (v == target) break;
        int v2 = v/2;
        int v3 = v*3;
        if (prev[v2] == 0) {q.push(v2); prev[v2] = v;}
        if (prev[v3] == 0) {q.push(v3); prev[v3] = v;}
    }
    if (v == target) showSolution(v, prev);
}
```

- Queue makes ordering of how we pick a state to enumerate
- From top to bottom and left to right

Display Solution

- Trace back "prev"

X	prev[x]
0	1
1	-1
2	4
3	1
4	9
6	2
9	3
12	4
13	27
27	9
36	12
39	13
40	81



showSolution

```cpp
void showSolution(int v, map<int,int>& prev) {
    string out = "";
    while(prev[v] != -1) {
        if (prev[v] * 3 == v) {
            out = "x3" + out;
        } else {
            out = "/2" + out;
        }
        v = prev[v];
    }
    out = "1" + out;
    cout << out << endl;
}
```

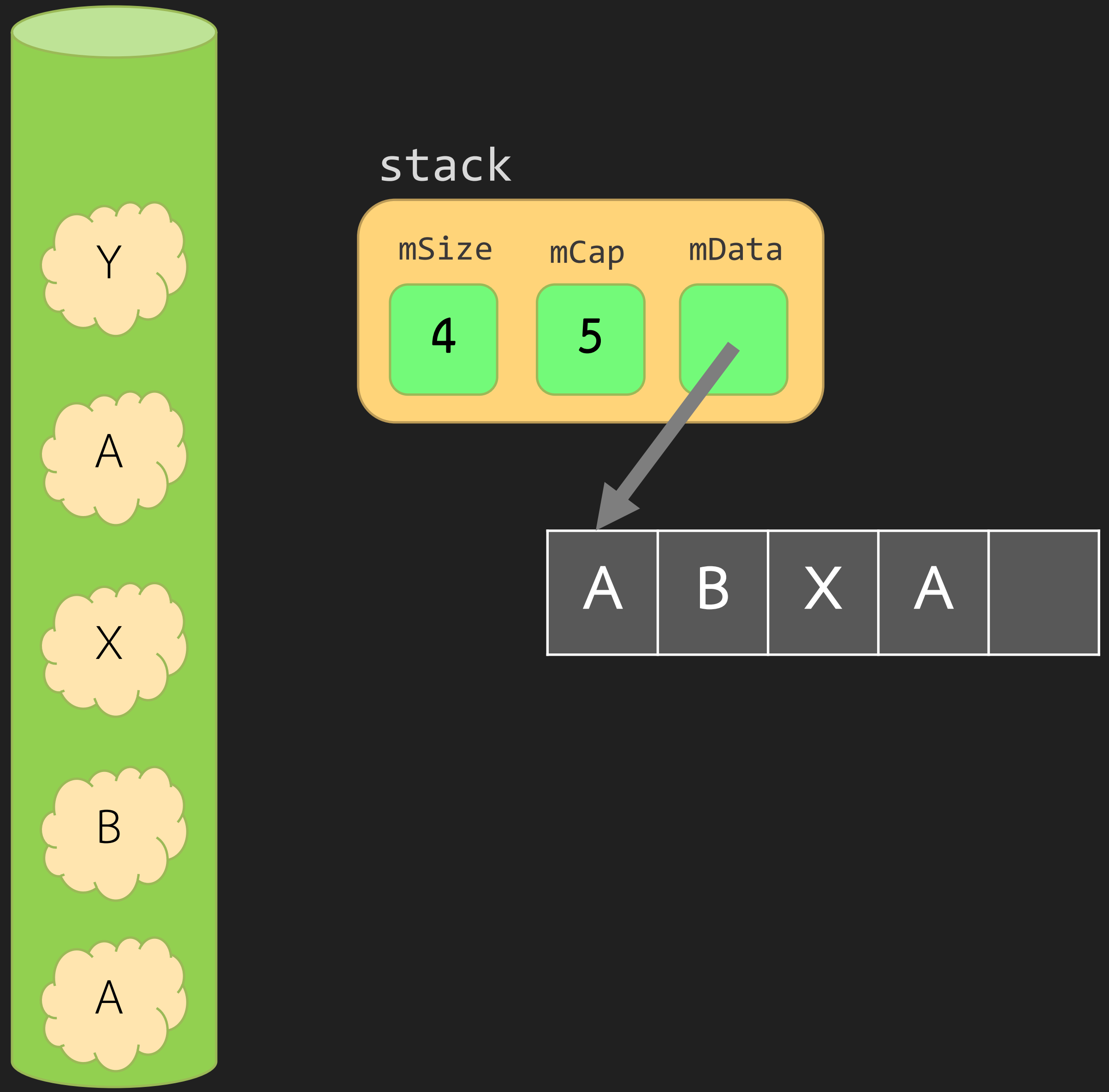
CP::stack

Intro

- Now we will create less complex data structure CP::stack
- Just like a vector without iterator, insert, erase, resize, at and operator[ ]
  - Add top() which is just a shorthand of looking at the last element
- That’s it, really

Key Idea

- The data is stored in the same way as a vector
    - The first element of mData is the bottom of stack while the last element is the top of stack
- We just take vector.h and remove unnecessary function



stack.h

```
namespace CP {
  template <typename T>
  class stack
  {
   protected:
     T *mData;
     size_t mCap;
     size_t mSize;
     void expand(size_t capacity)  {...}
     void ensureCapacity(size_t capacity)  {...}
   public:
     //------------- constructor ----------
     stack(const stack<T>& a) {...}
     stack()  {...}
     stack<T>& operator= {...}
     ~stack()  {...}
     //-------- capacity function ---------
     bool empty() const {...}
     size_t size() const {...}
     //------------ access ---------------
     const T& top() const {...}
     //------------ modifier -------------
     void push(const T& element) {...}
     void pop() {...}
  };
}
```

Same as vector

```
const T& top() const{
  return mData[mSize-1];
}
```

This is push_back

This is pop_back

Speed of each operation

- All read operation always take constant time
  - size(), top() simply return something that is directly accessible
- All modify operation also take constant time
  - push() is constant on average (same as push_back of vector)
  - pop() is always constant

Stack By Vector

- Instead of writing our own function, there is another way to write a stack
- We simply use vector as our sole data member
- Benefit: code reuse
- Drawback: almost none except that we need one more layer of function call

```cpp
namespace CP {
  template <typename T>
  class stack
  {
    protected:
      vector<T> v;
    public:
      // default constructor
      stack() : v() { }
      //------------- capacity function -------------------
      bool empty() const          {    return v.empty(); }
      size_t size() const         {     return v.size(); }
      //----------------- access -------------------------
      const T& top() const       { return v[v.size()-1]; }
      //----------------- modifier -----------------------
      void push(const T& element) { v.push_back(element); }
      void pop()                          { v.pop_back(); }
  };
}
```

CP::queue

Will the circle be unbroken?

Nattee Niparnan

Intro

- Queue, unlike stack, require more sophisticated technique to achieve fast performance
- We start by writing a simple class that just work (slowly)
- Then we try to improve it

Key Idea

- Just like stack, we will use the same format as vector, using dynamic array to store data
- However, we have to somehow manage how we works with front() and back() of the queue

v0.1 simple implementation of queue

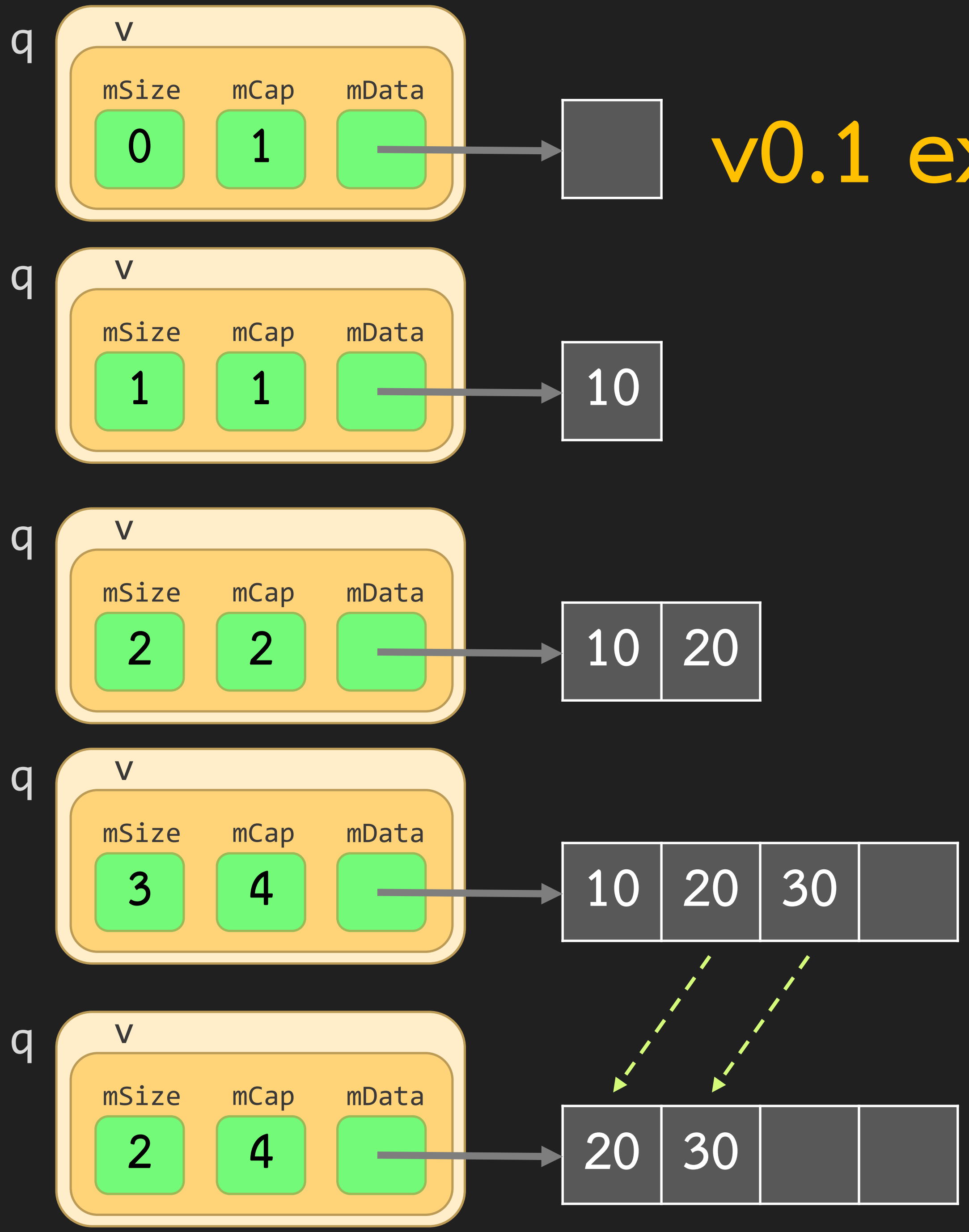
- To illustrate this idea, we will use a vector as our data member
- `push(e)` is simply `v.push_back(e)`, this is fast
- `front()` is `v[0]`, `back()` is `v[v.size()-1]`, this is also fast
- `pop()` is `v.erase(v.begin())`, this is slow (always propotional to `v.size()`)
  - Unlike std::queue which has very fast pop()

```
namespace CP {
  template <typename T>
  class queue {
    protected:
      std::vector<T> v;
    public:
      //------------- capacity function ------------------
      queue() : v() {}
      //------------- capacity function ------------------
      bool empty() const                  { return v.empty();}
      size_t size() const                  { return v.size();}
      //---------------- access ------------------------
      const T& front() const                  { return v[0];}
      const T& back() const          { return v[v.size()-1];}
      //---------------- modifier -----------------------
      void push(const T& element)  { v.push_back(element);}
      void pop()                      { v.erase(v.begin());}

  };
}
```

v0.1 example

```cpp
CP::queue<int> q;
q.push(10);
q.push(20);
q.push(30);
q.pop();
```



v0.2 faster queue

- Add more data member mFront, initialized as 0
- `push(e)` is simply `v.push_back(e)`, this is fast
- `front()` is `v[mFront]`, `back()` is `v[v.size()-1]`, this is also fast
- `pop()` is `mFront++`, this is fast
    - However, we don't really remove anything when pop

```
#include <vector>

namespace CP {
  template <typename T>
  class queue
  {
    protected:
      std::vector<T> v;
      int mFront;
    public:
      //------------- capacity function -------------------
      queue() : v(), mFront() {}
      //------------- capacity function -------------------
      bool empty() const                     { return v.empty();}
      size_t size() const          { return v.size()- mFront;}
      //---------------- access ------------------------
      const T& front() const                  { return v[mFront];}
      const T& back() const             { return v[v.size()-1];}
      //----------------- modifier ------------------------
      void push(const T& element)  { v.push_back(element);}
      void pop()                                   { mFront++;}
  };
}
```

v0.2 example

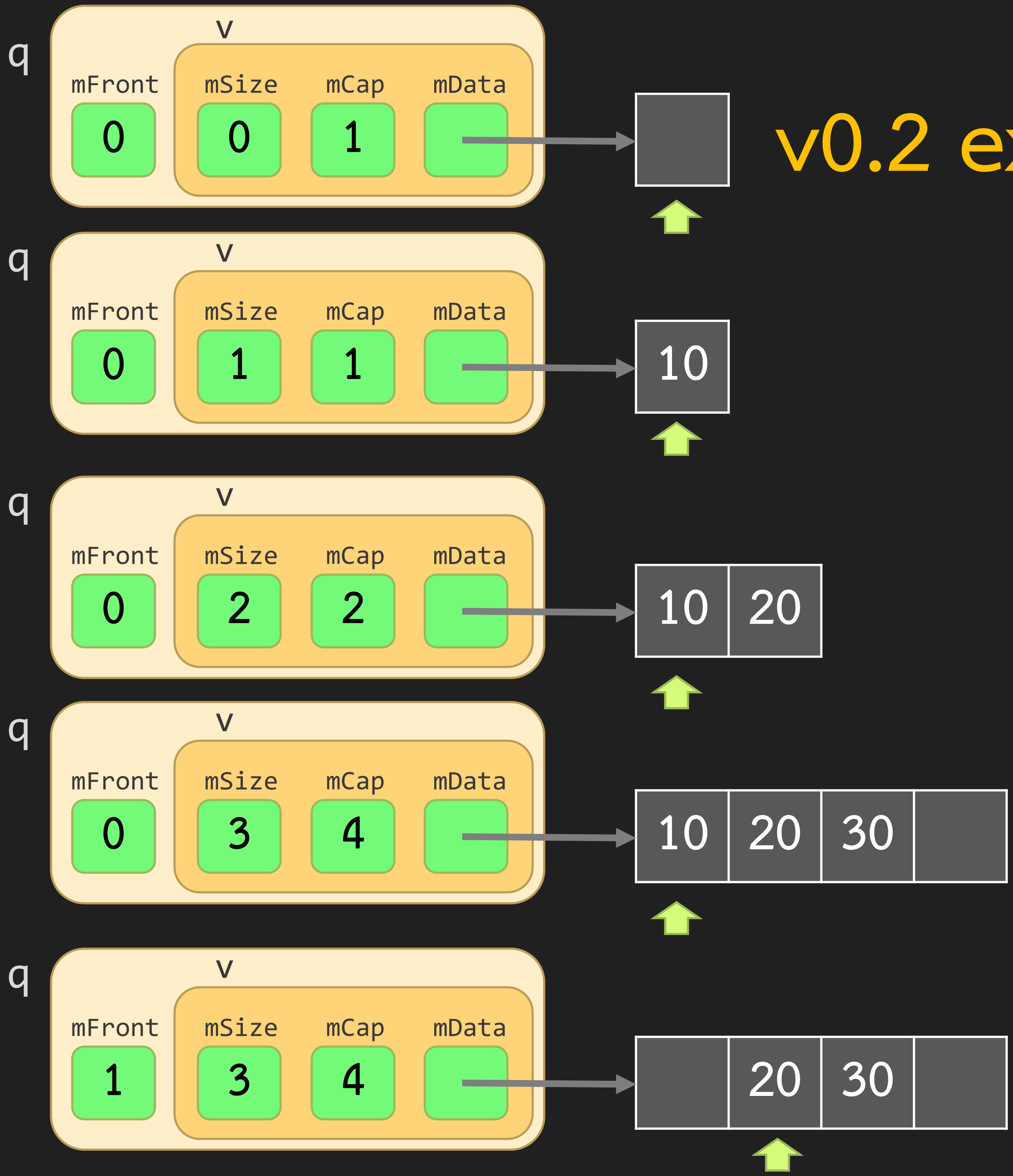
```
CP::queue<int> q;
```

```
q.push(10);
```

```
q.push(20);
```

```
q.push(30);
```

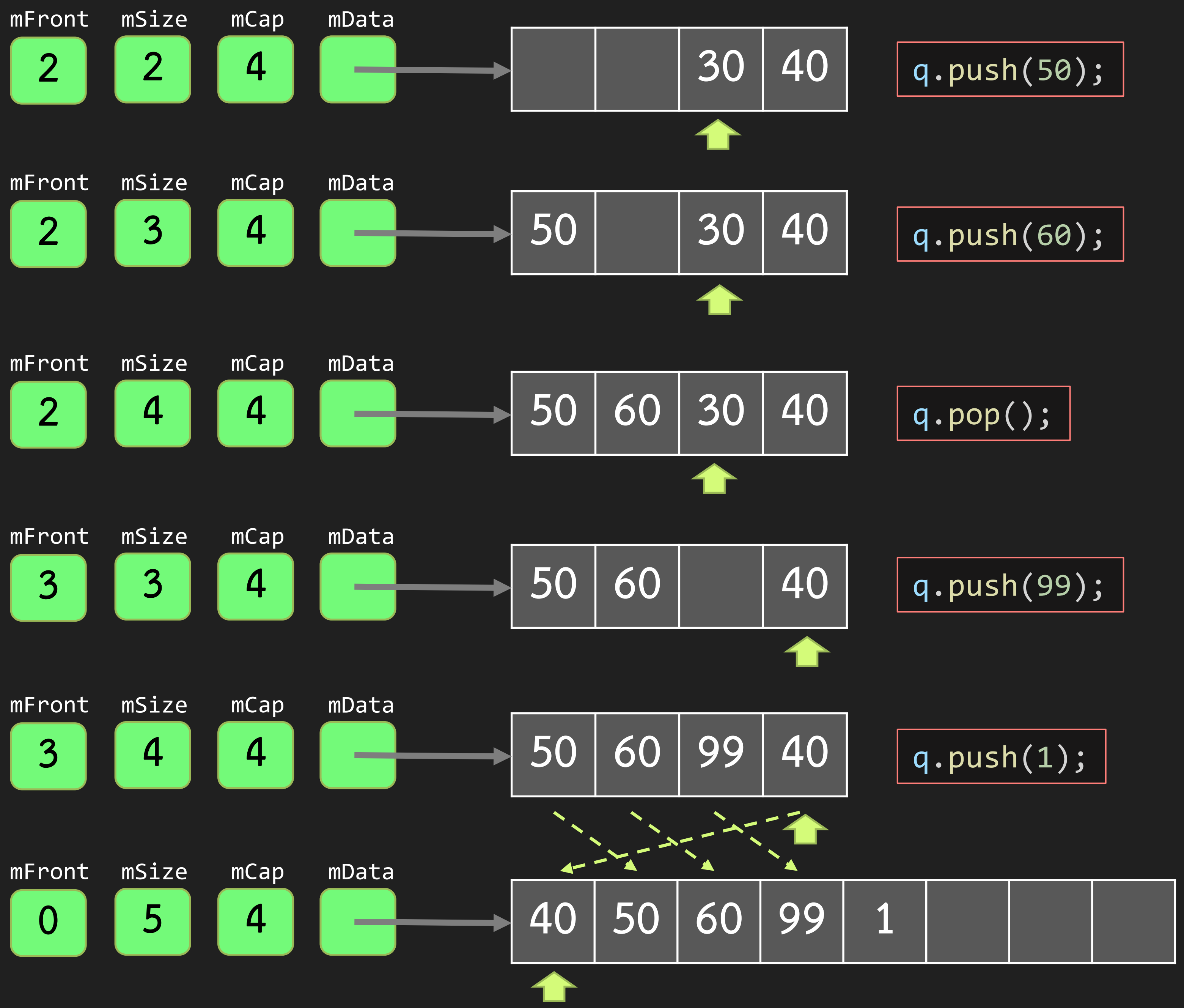
```
q.pop();
```



Problem with v0.2

- Fast but use too many space
- Queue grows according to how many time push is called
  - regardless of how many pop is called
- The data stored in the vector can be much larger than the actual data in the queue
- Does not really work in real world

```cpp
for (int i = 0;i < 1000000;i++) {
  q.push(i);
  q.pop();
}
std::cout << q.size() << std::endl;
```

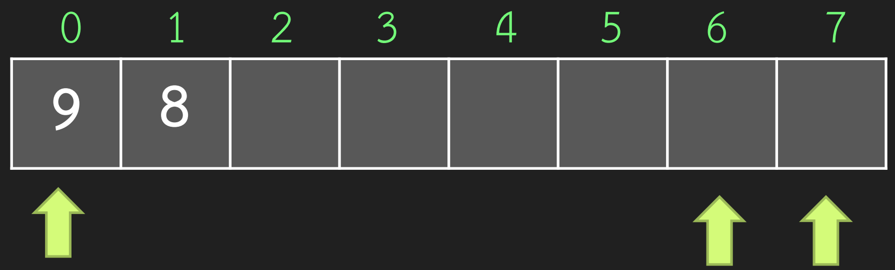


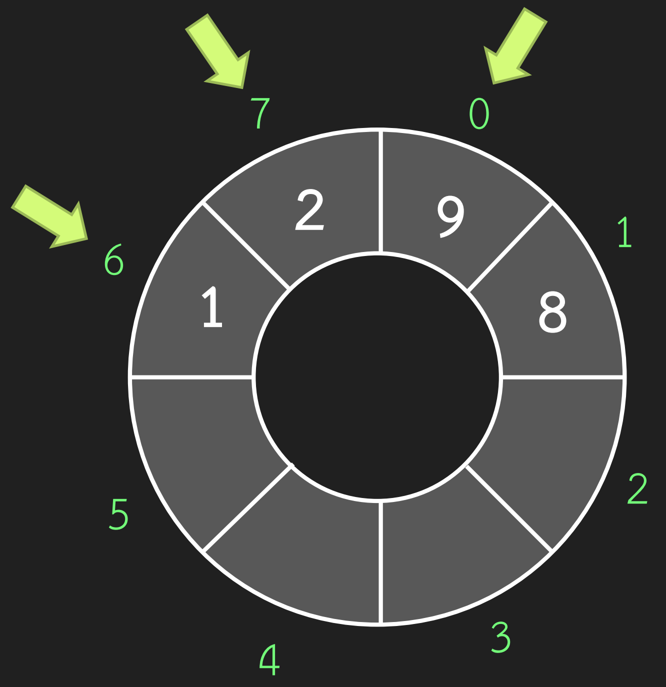
Final Idea

- We take v0.2 and reuse the area at the beginning of mData
  - Expand when necessary
  - Re-arrange when expand

Circular Queue

- We can think of mData to be circular
  - End of the last element of the mData is connected to the first element
- Consider i^{th} element
  - the next element is `(i+1) % mCap`
  - The previous element is `(i-1+mCap) % mCap`
  - Next k element is `(i+k) % mCap`





queue.h

```
namespace CP {
  template <typename T>
  class queue
  {
    protected:
      T *mData;
      size_t mCap;
      size_t mSize;
      size_t mFront;
      void expand(size_t capacity) {...}
      void ensureCapacity(size_t capacity) {...}
    public:
      //-------------- constructor --------------
      queue(const queue<T>& a) {...}
      queue() {...}
      queue<T>& operator=(queue<T> other) {...}
      ~queue() {...}
      //------------- capacity function ---------
      bool empty() const {...}
      size_t size() const {...}
      //----------------- access ----------------
      const T& front() const  {...}
      const T& back() const  {...}
      //----------------- modifier --------------
      void push(const T& element)  {...}
      void pop()  {...}
  };
}
```

Additional data member mFront

Almost the same but have to take care of mFront

Same as vector

Circular queue implementation

Ctor, Dtor, copy

```
template <typename T>
class queue {
  protected:
    T *mData;  size_t mCap;  size_t mSize;   size_t mFront;
  public:
    // default constructor
    queue() : mData(new T[1]()), mCap(1),
              mSize(0), mFront(0) { }
    // copy constructor
    queue(const queue<T>& a) : mData(new T[a.mCap]()), mCap( a.mCap ),
                               mSize( a.mSize ), mFront( a.mFront ) {
      for (size_t i = 0; i < a.mCap;i++) {
        mData[i] = a.mData[i];
      }
    }
    // copy assignment operator
    queue<T>& operator=(queue<T> other) {
      using std::swap;
      swap(mSize,other.mSize);
      swap(mCap,other.mCap);
      swap(mData,other.mData);
      swap(mFront,other.mFront);
      return *this;
    }
    ~queue() {
      delete [] mData;
    }
};
```

List initialization

Need to copy entire mData (not just mSize)

Also swap mFront

same

- Dtor is the same
- ctor also have to initialize mFront
- Copy also have to copy mFront

front(), back(), pop()

```
template <typename T>
class queue {
  protected:
    T *mData;
    size_t mCap;
    size_t mSize;
    size_t mFront;
  public:
    //----------------- access -----------------
    const T& front() const {
      return mData[mFront];
    }
    const T& back() const {
      return mData[(mFront + mSize - 1) % mCap];
    }
    //----------------- modify -----------------
    void pop() {
      mFront = (mFront + 1) % mCap;
      mSize--;
    }
};
```

- back = mFront + mSize -1
  - Also circular (by % mCap)
- pop = move mFront by 1
  - Also circular
  - Also change size

push, expand

- push add data to `(mFront+mSize) % mCap`
  - The space just after `back()`
- Expand re-pack the `mData` so that `mFront` is 0
- `ensureCapacity` is the same

```
template <typename T>
class queue {
  protected:
    T *mData;
    size_t mCap;
    size_t mSize;
    size_t mFront;
    void expand(size_t capacity) {
      T *arr = new T[capacity]();
      for (size_t i = 0;i < mSize;i++) {
        arr[i] = mData[(mFront + i) % mCap];
      }
      delete [] mData;
      mData = arr;
      mCap = capacity;
      mFront = 0;
    }
    void ensureCapacity(size_t capacity) {
      if (capacity > mCap) {
        size_t s = (capacity > 2 * mCap) ? capacity : 2 * mCap;
        expand(s);
      }
    }
  public:
    void push(const T& element) {
      ensureCapacity(mSize+1);
      mData[(mFront + mSize) % mCap] = element;
      mSize++;
    }
};
```

Analysis

- All access, modification is fast (constant time)
- Space is re-used
  - It is not shrunk when mSize reduce
  - Space is not more than double of maximum mSize during its lifetime

Exercise

- We implement circular queue by maintain mFront and use circular logic (% mCap) to calculate the position of back of the queue
  - Can we maintain mBack instead?
  - Can we maintain both mFront and mBack but not mSize?
- How about mCap, if we know mFront, mSize, mBack, can we calculate mCap?

mFront	mSize	mBack	front()	back()	size()
YES	YES	No	v[mFront]	v[(mFront + mSize - 1) % mCap]	mSize
No	YES	YES	????	v[mBack]	mSize
YES	No	YES	v[mFront]	v[mBack]	????

Now, meet deque

- Can you modify queue to include
  - push_front(), add to the front of the queue
  - pop_back(), remove from back of the queue
- All operation should still be constant time