Do Now Exercise

To prepare you for the lecture today, please do the following exercise.

Write the asymptotic running time of finding an item

performed on each of the data structures that we have learned so far.

COMP15: Data Structures

Week 9, Summer 2019

Admin

T8: **clang++, valgrind, make**

Due by 6pm on Wednesday, July 24

(a quick demo)

Ref: make http://www.gnu.org/software/make/manual/make.html

(heads-up)

T9: Pick commands that you are interested in, and prepare the in-class presentation.

(heads-up)
T9: Presentation Sign-Up sheet is on Canvas

P4: **Course Registration System**

Project Due by 6pm on Sunday, July 21

"“Make it work, then make it beautiful, then if you really, really have to, make it fast.”
...
“90 percent of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful!”"

- Joe Armstrong

from Erlang and OTP in Action (Chapter 14.)
by Richard Carlsson, Eric Merritt, Martin Logan

Any questions about P4?

File IO?

(Slide from Week 8)

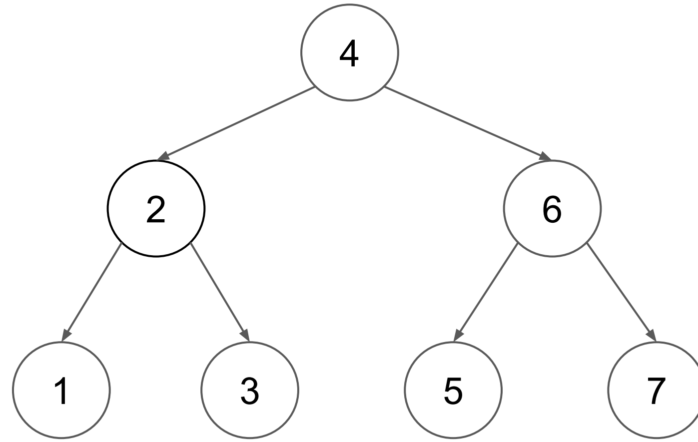
Self-balancing binary search trees

- AVL tree
- Red-black tree
- (and more)

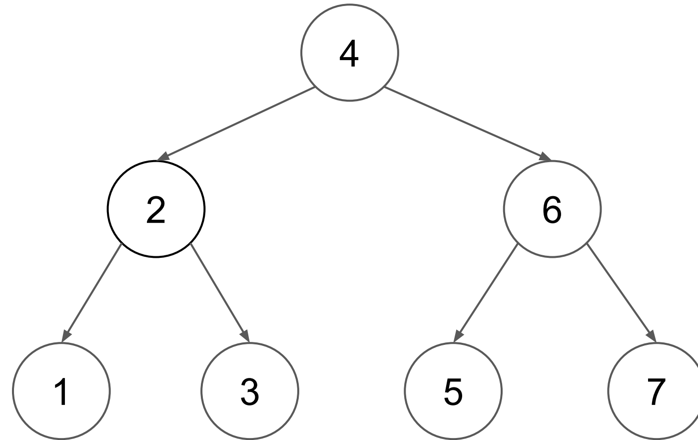
Self-adjusting binary search trees

- Splay tree

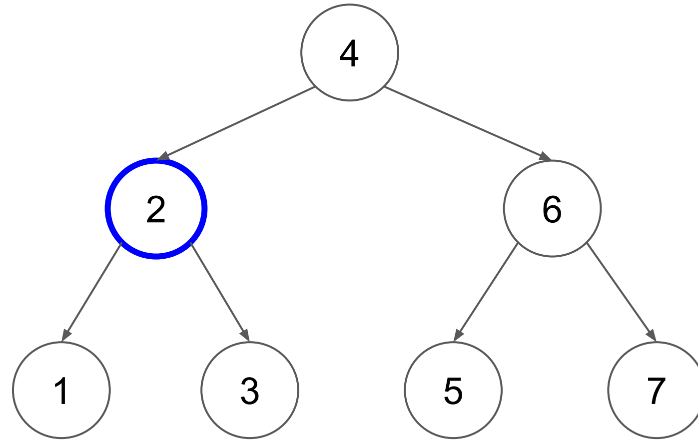
Splay tree


4
2
6
1
3
5
7

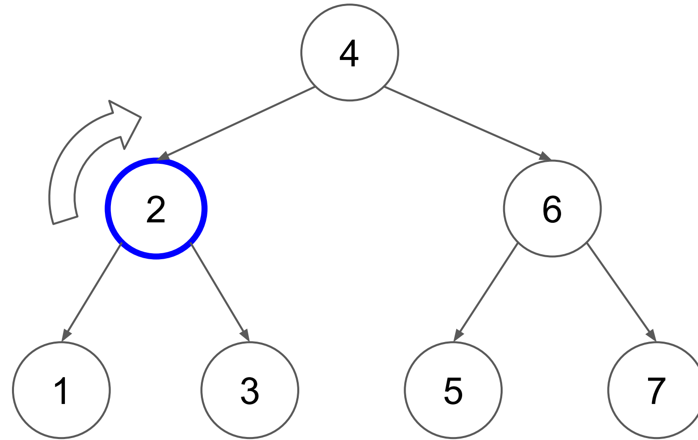
search(2);



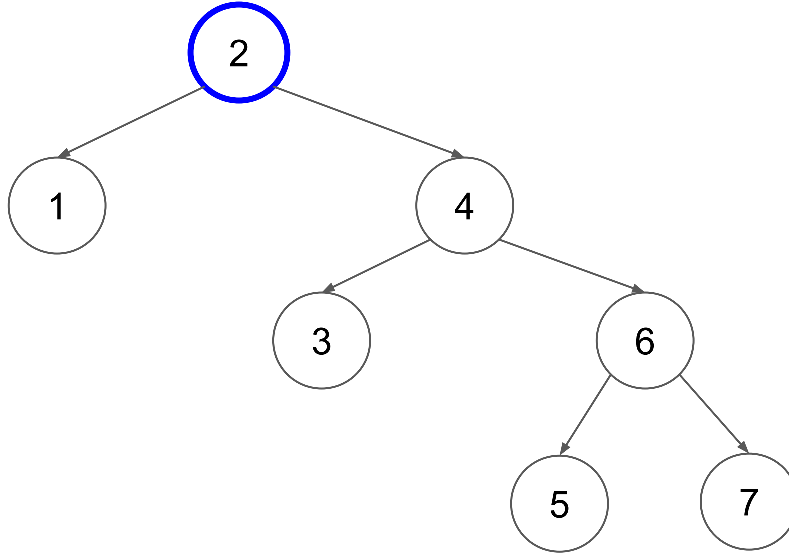
search(2);



search(2);



search(2);



Splaying

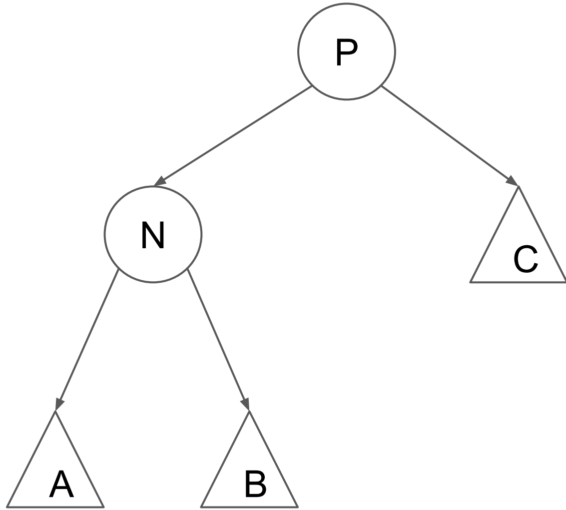
zig
zig-zig
zig-zag

(N)ode
(P)arent
(G)randparent

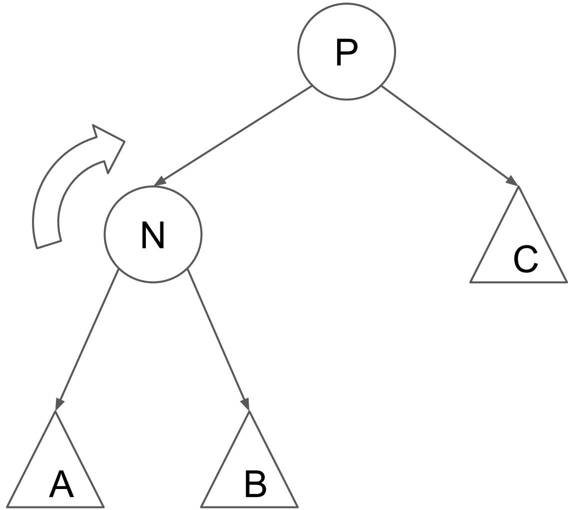
zig (R)

when N is the left of P, and P is the root.

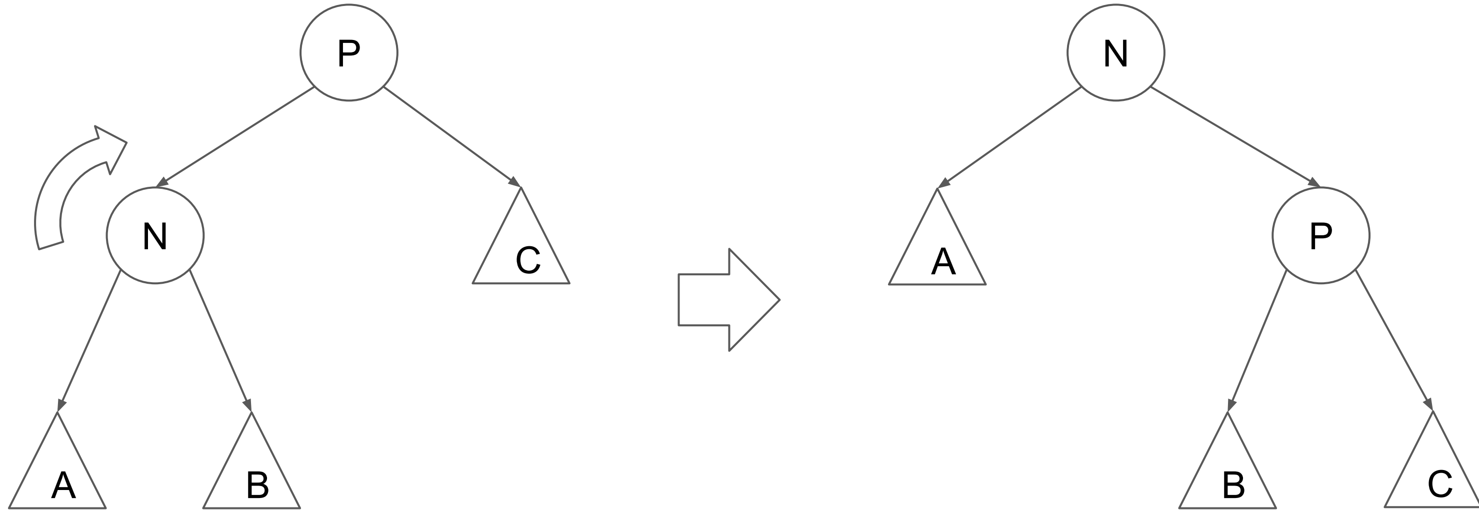
zig (R)



zig (R)



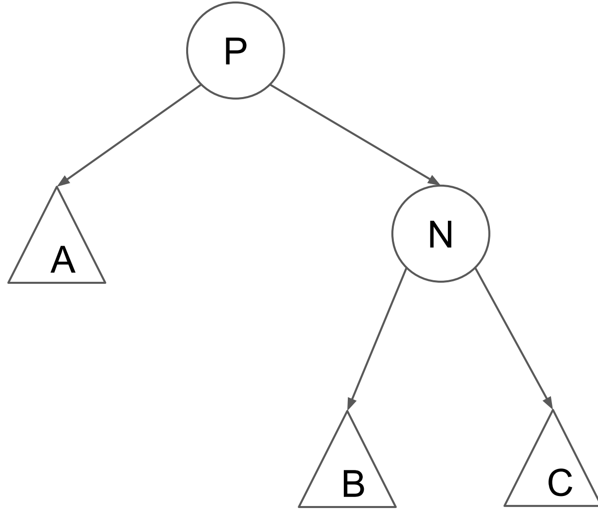
zig (R)



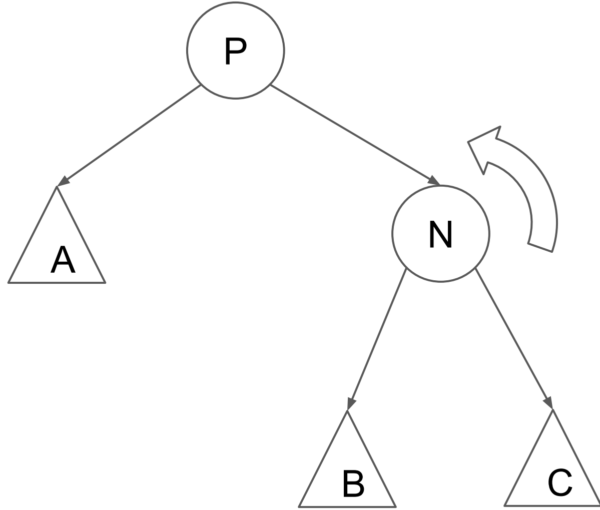
zig (L)

when N is the right of P, and P is the root.

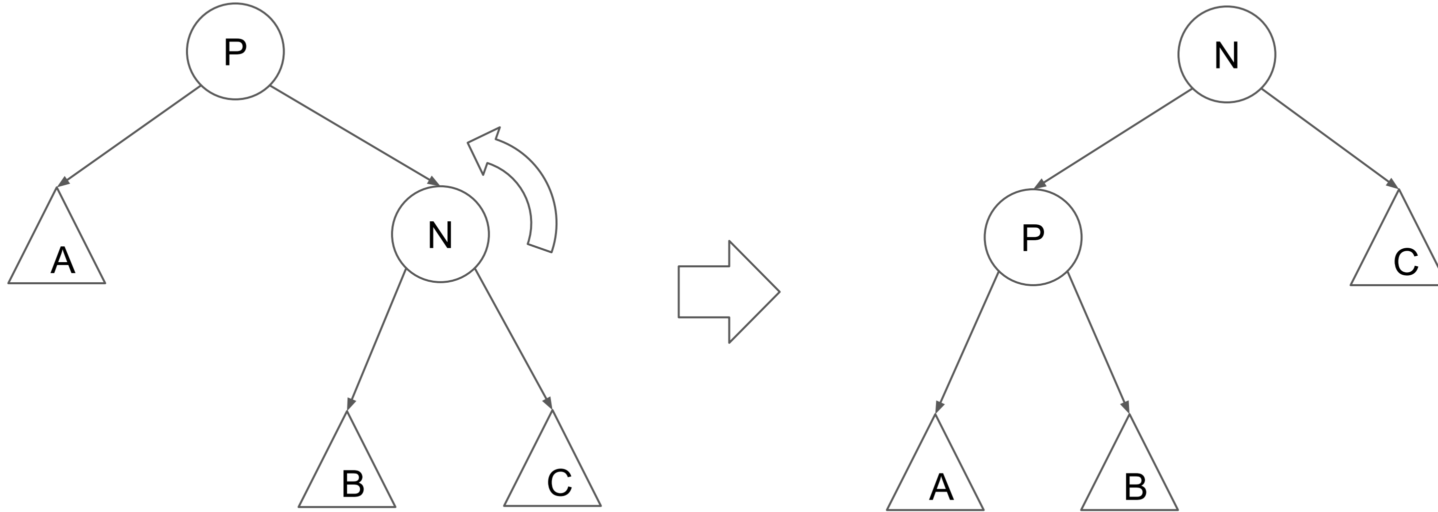
zig (L)



zig (L)



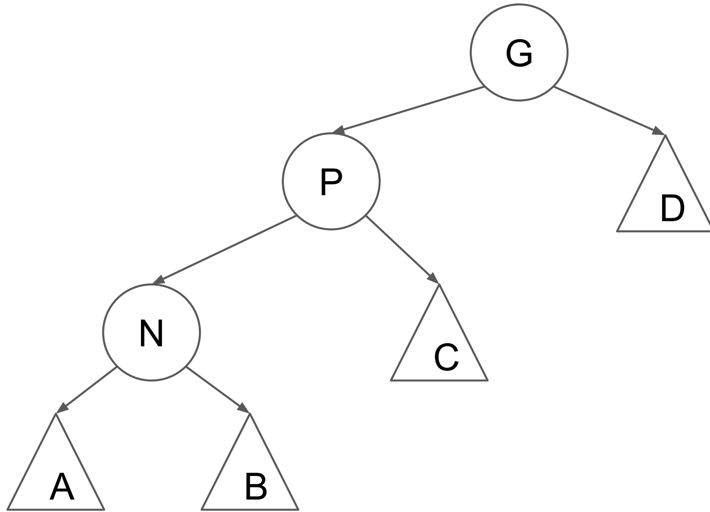
zig (L)



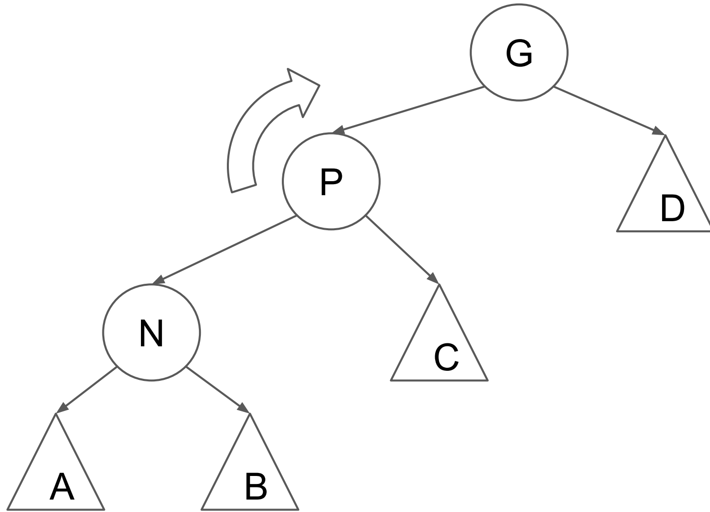
zig-zig (RR)

when N is the left of P, and P is the left of G.

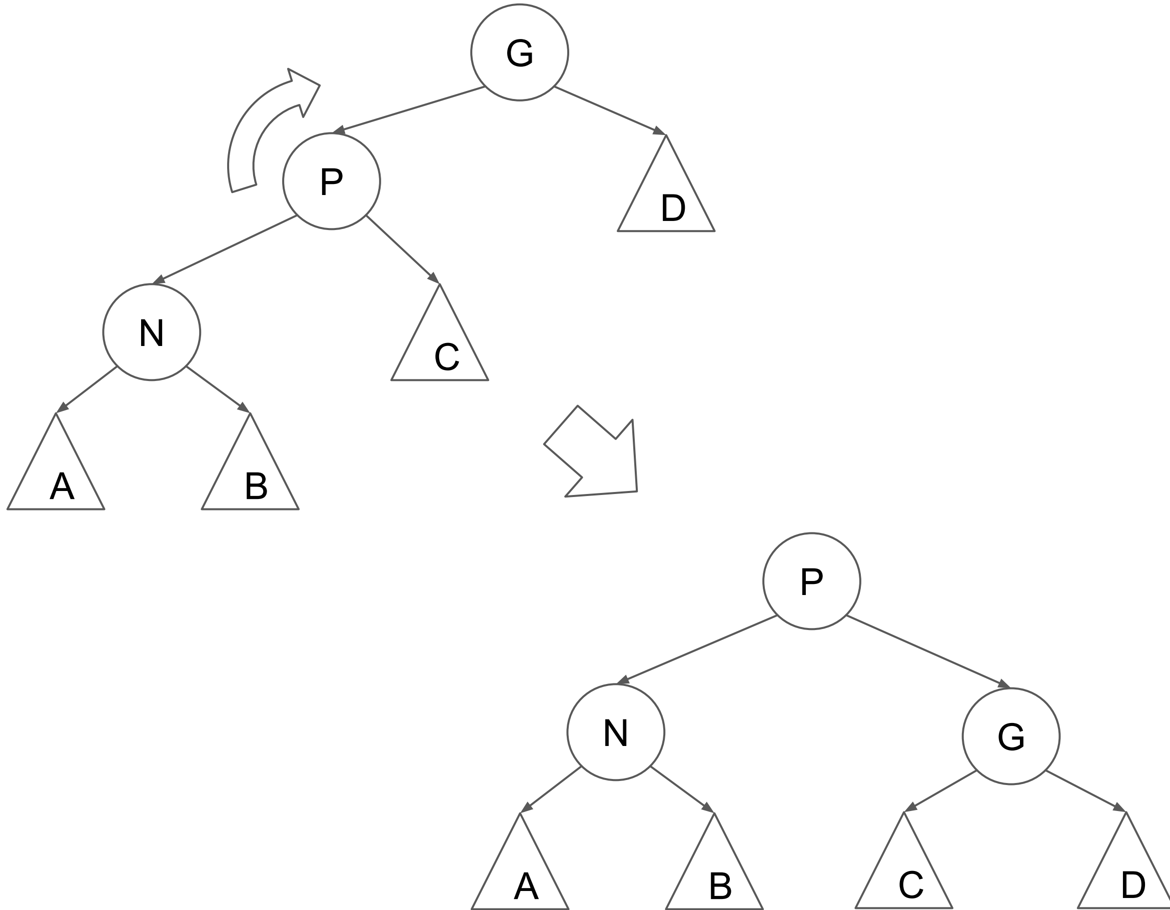
zig-zig (RR)



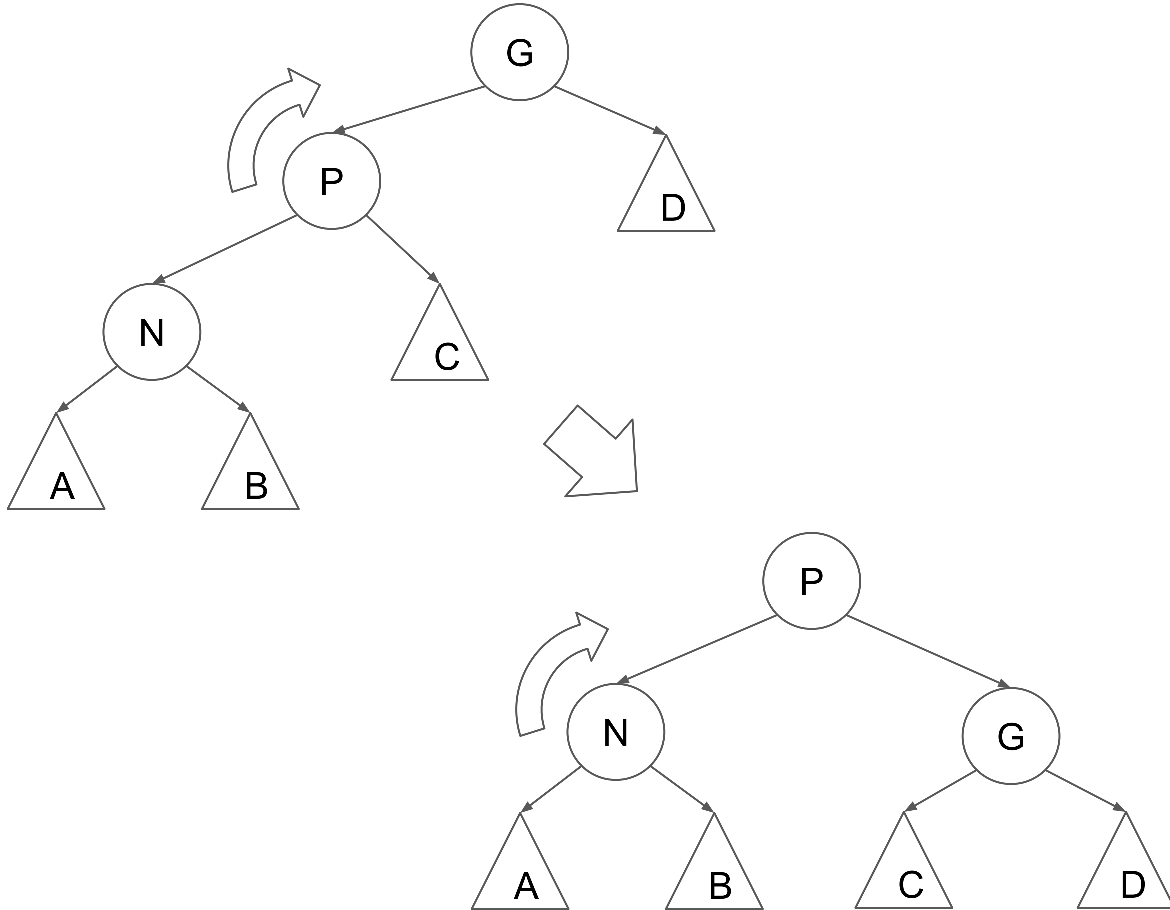
zig-zig (RR)



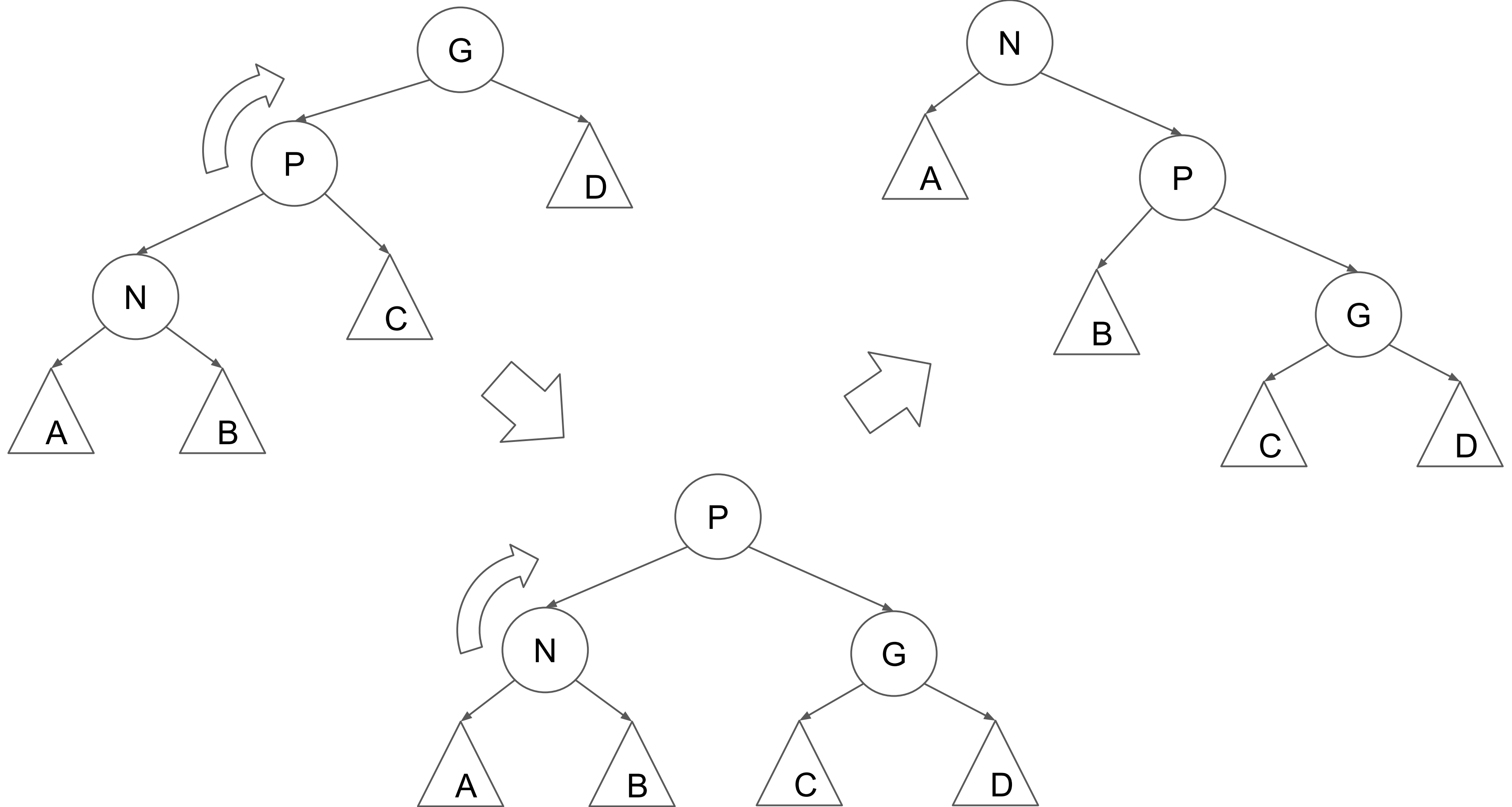
zig-zig (RR)



zig-zig (RR)



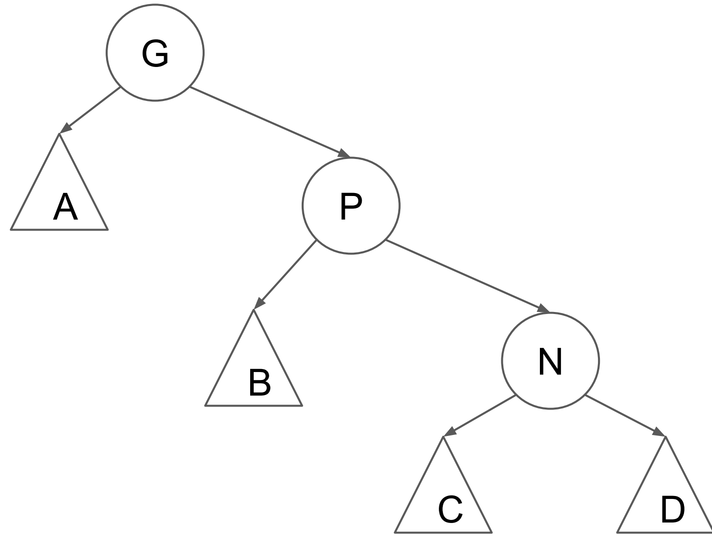
zig-zig (RR)



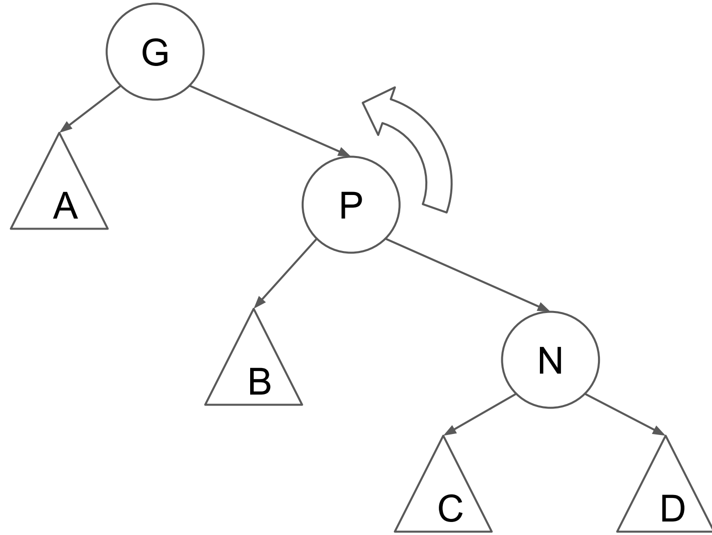
zig-zig (LL)

when N is the right of P, and P is the right of G.

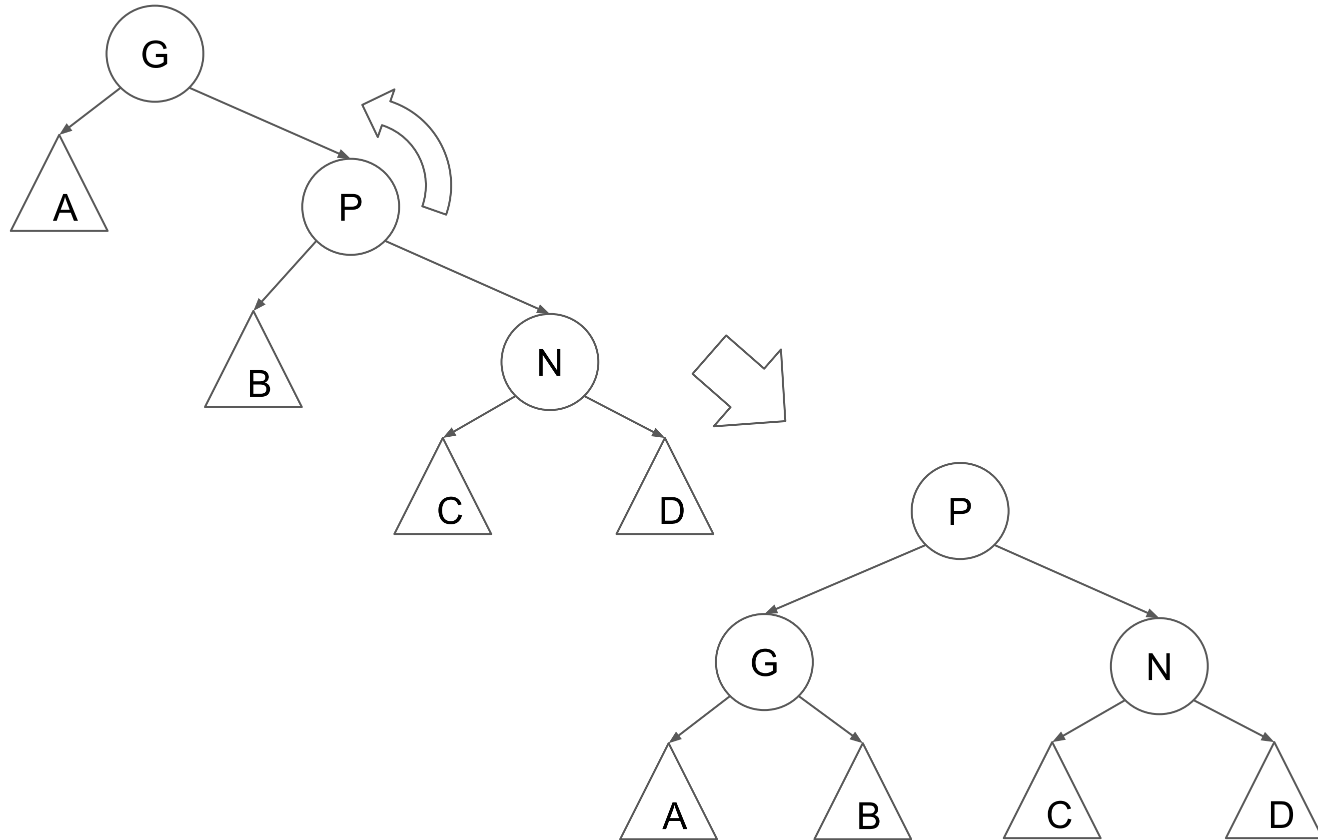
zig-zig (LL)



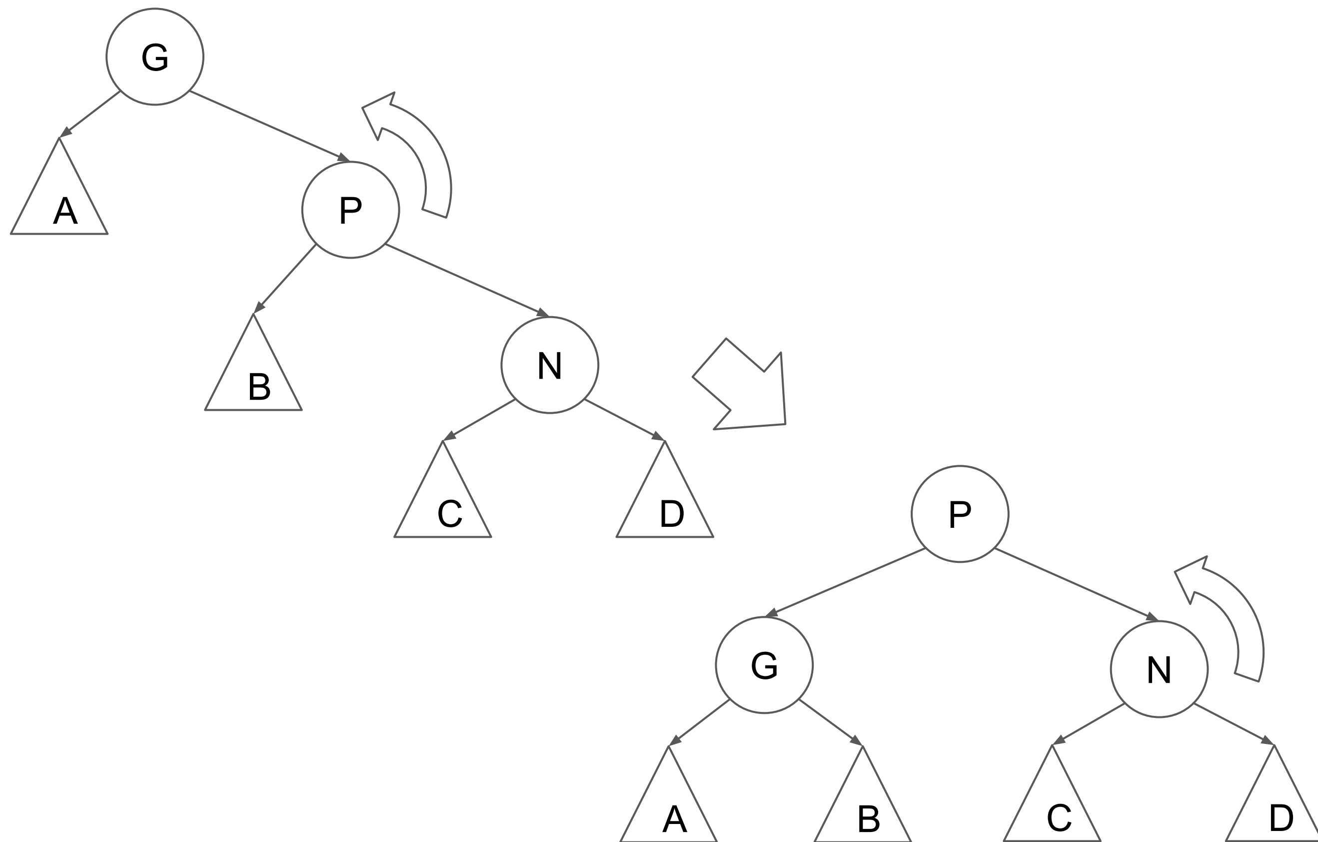
zig-zig (LL)



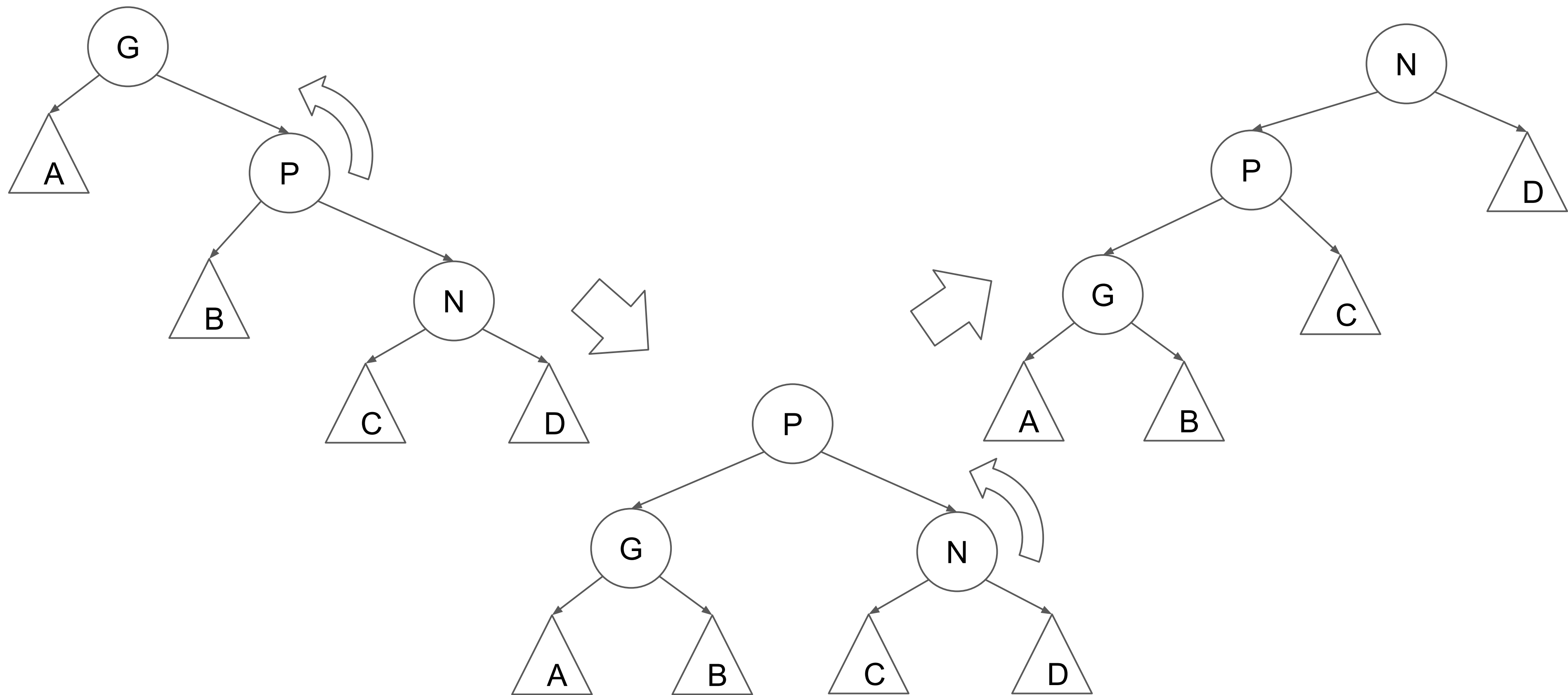
zig-zig (LL)



zig-zig (LL)



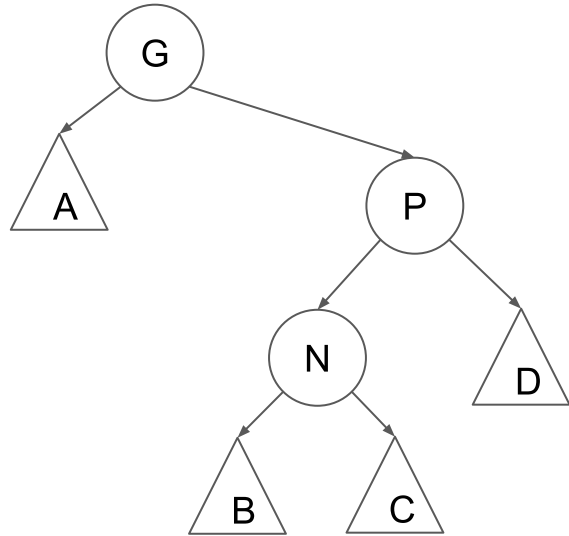
zig-zig (LL)



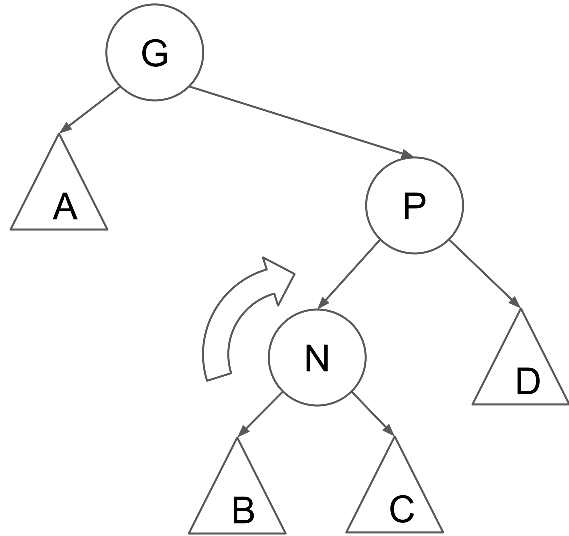
zig-zag (RL)

when N is the left of P, and P is the right of G.

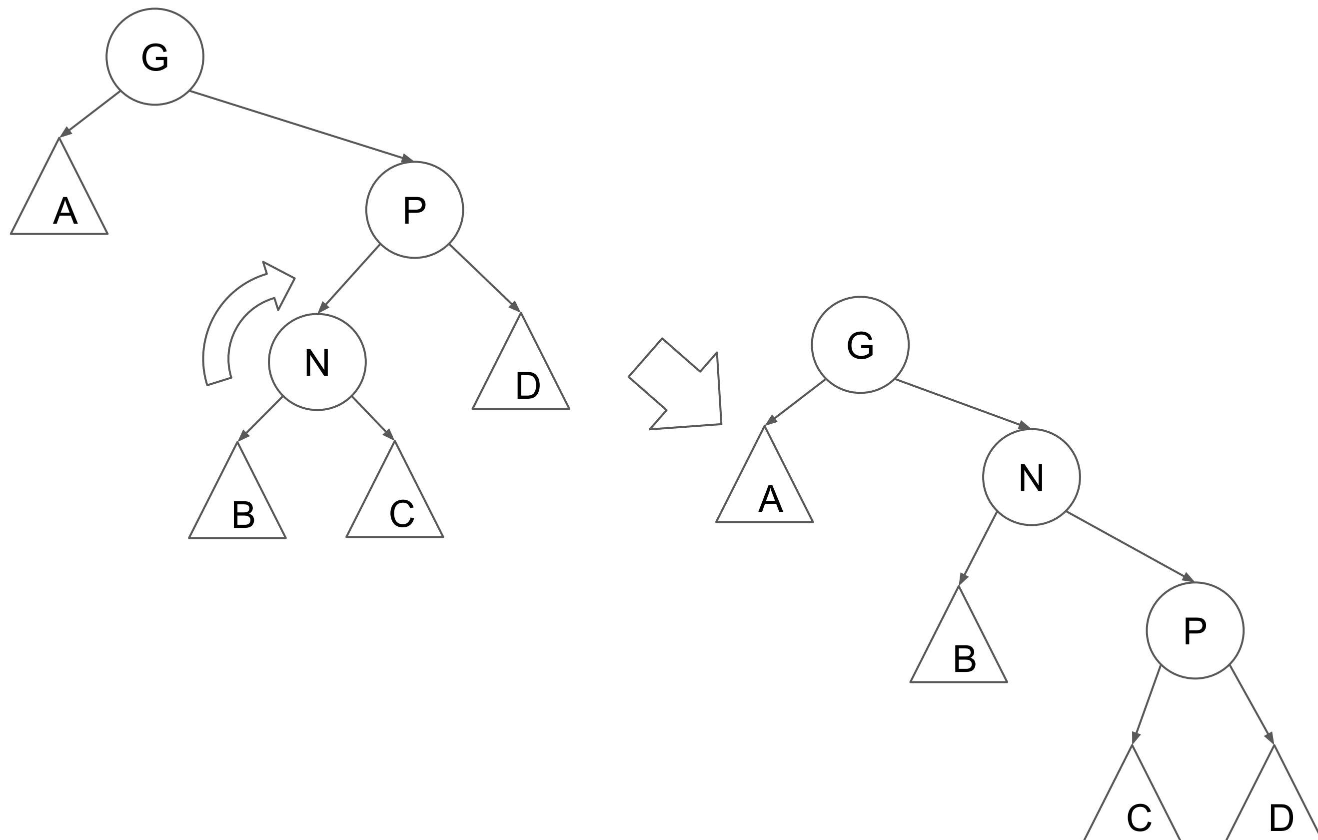
zig-zag (RL)



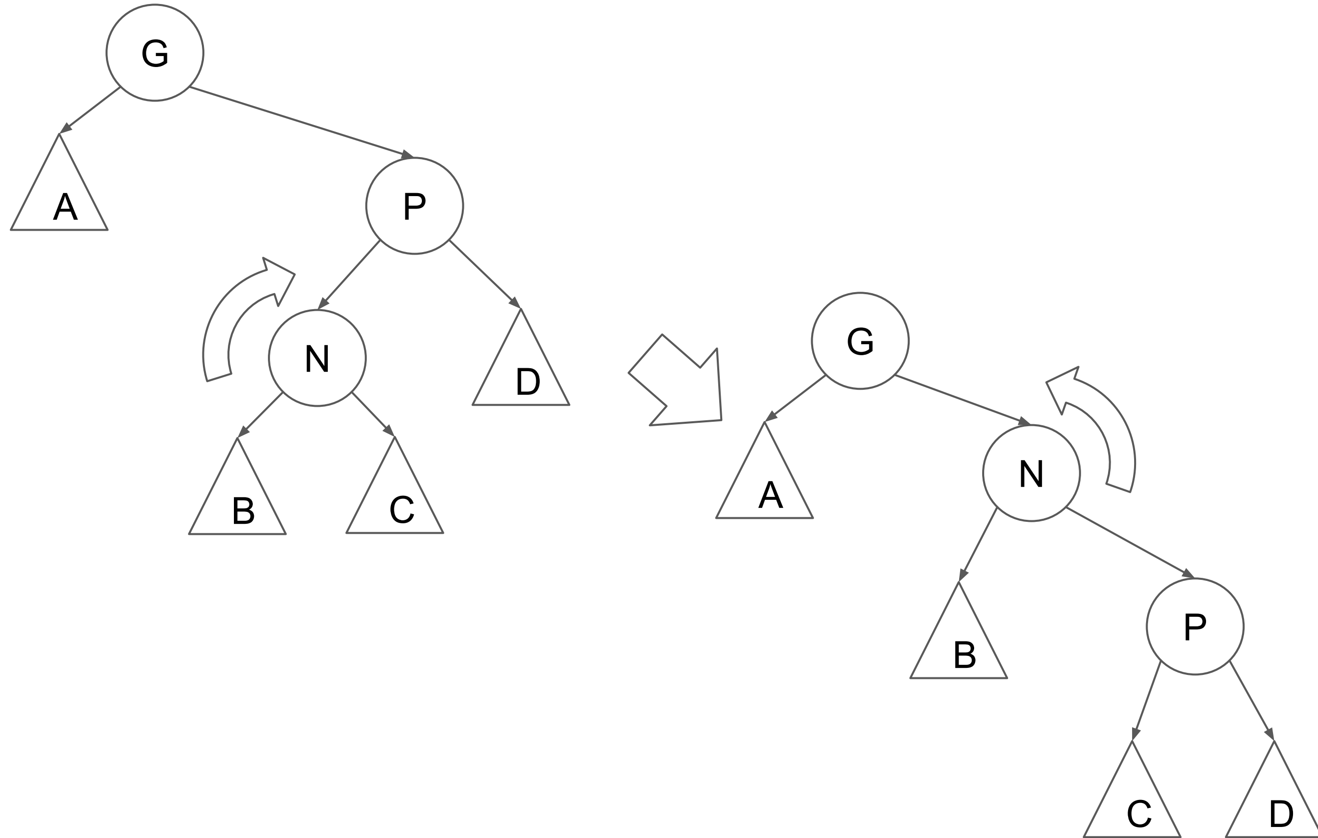
zig-zag (RL)



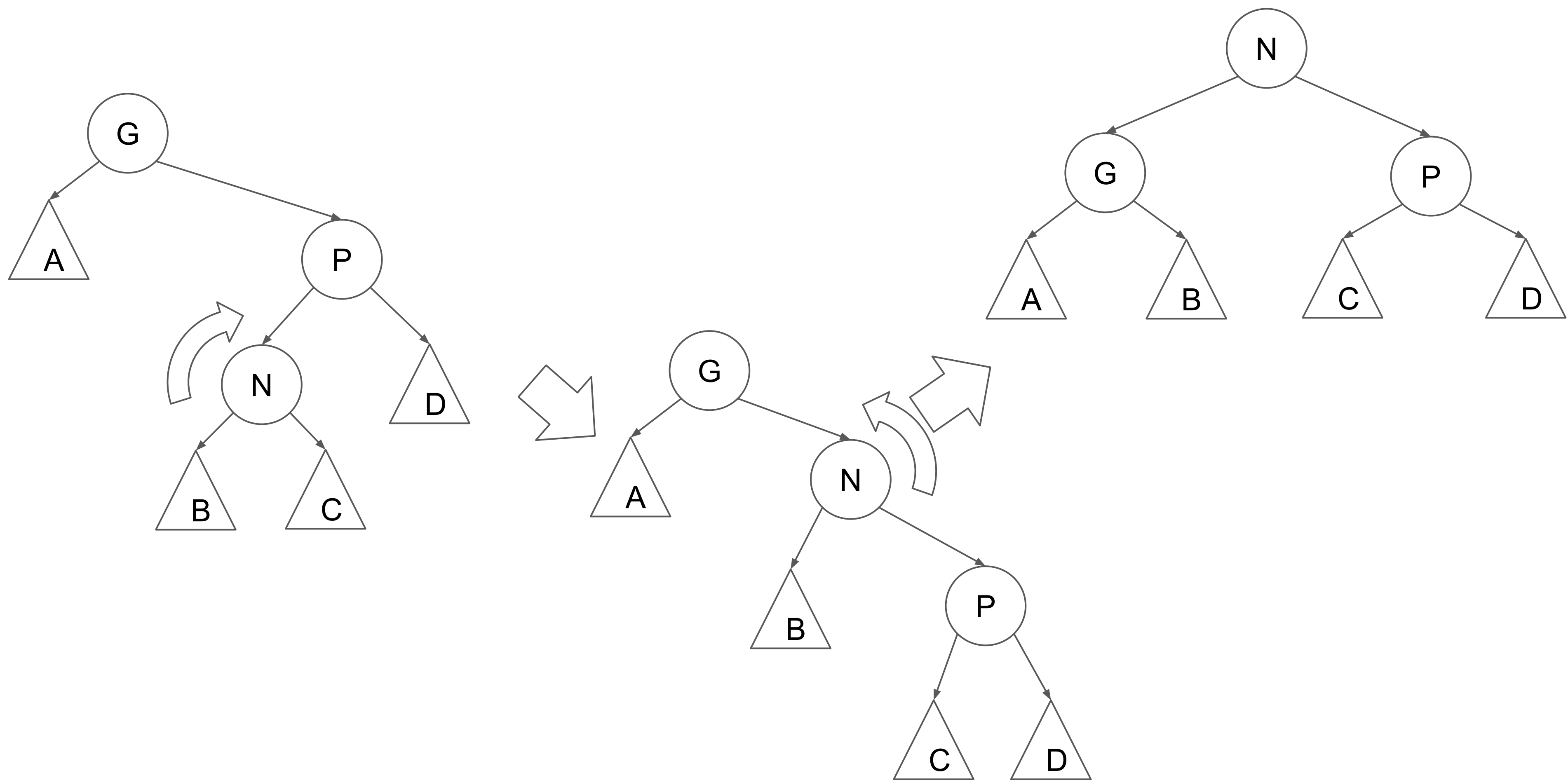
zig-zag (RL)



zig-zag (RL)



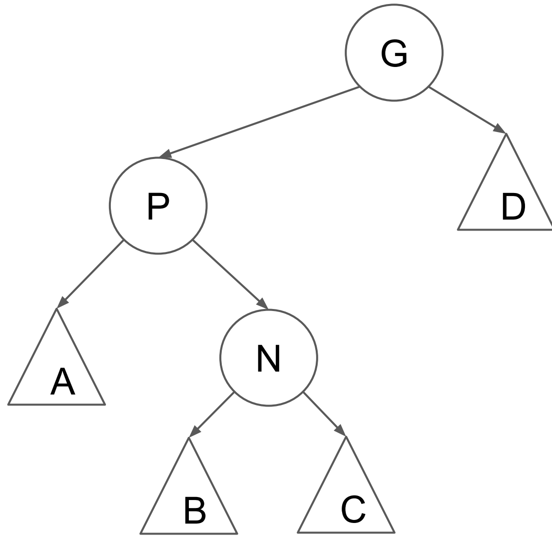
zig-zag (RL)



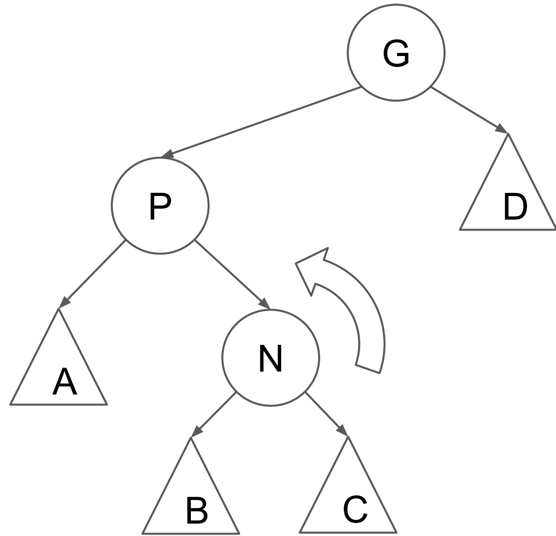
zig-zag (LR)

when N is the right of P, and P is the left of G.

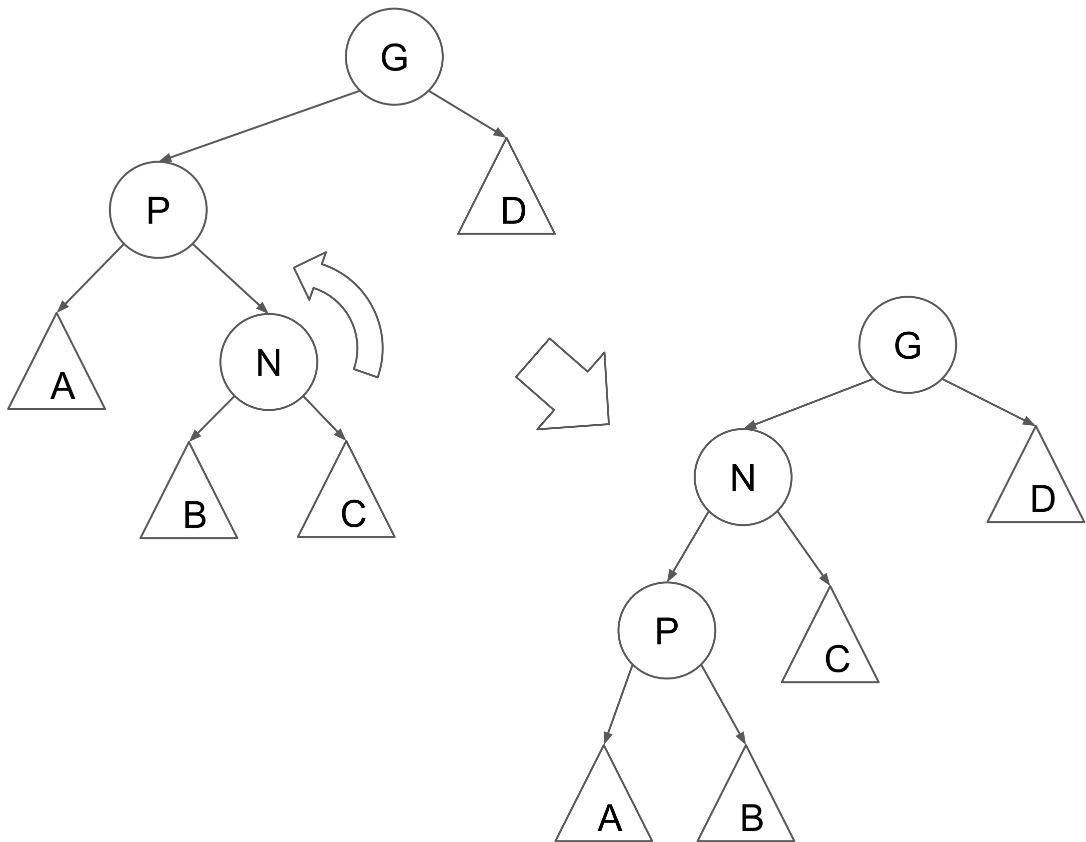
zig-zag (LR)



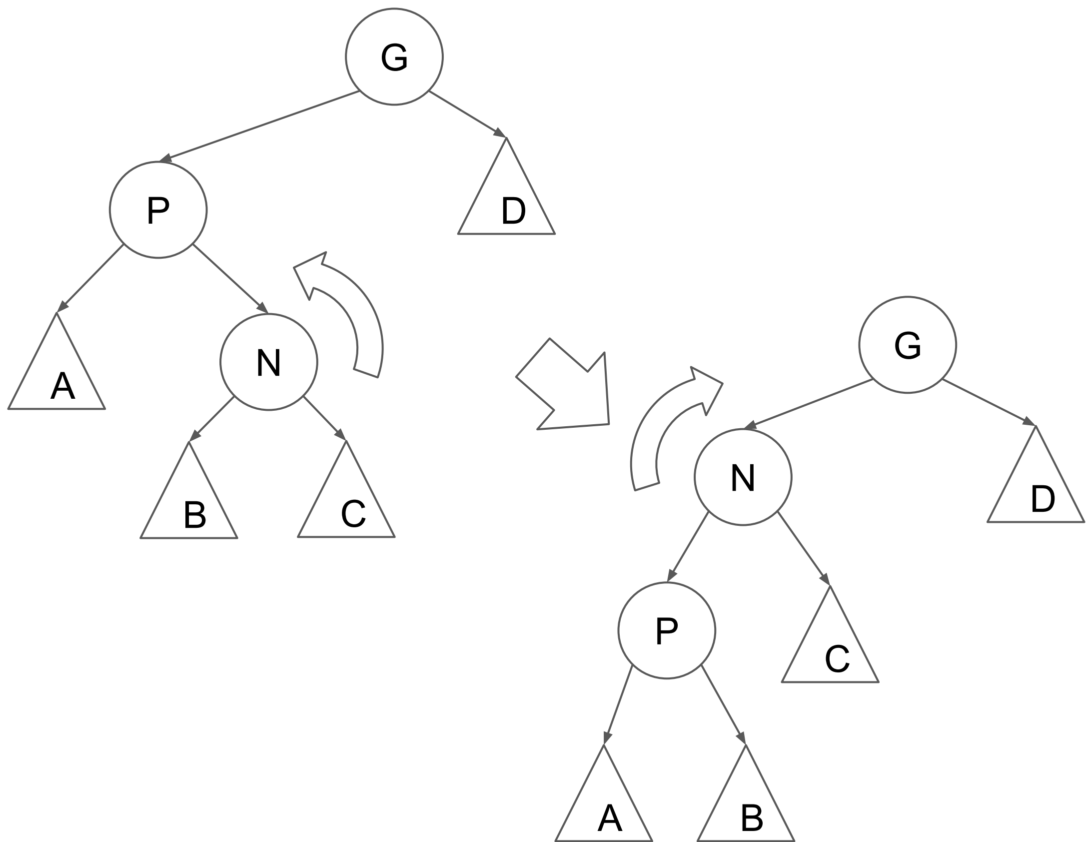
zig-zag (LR)



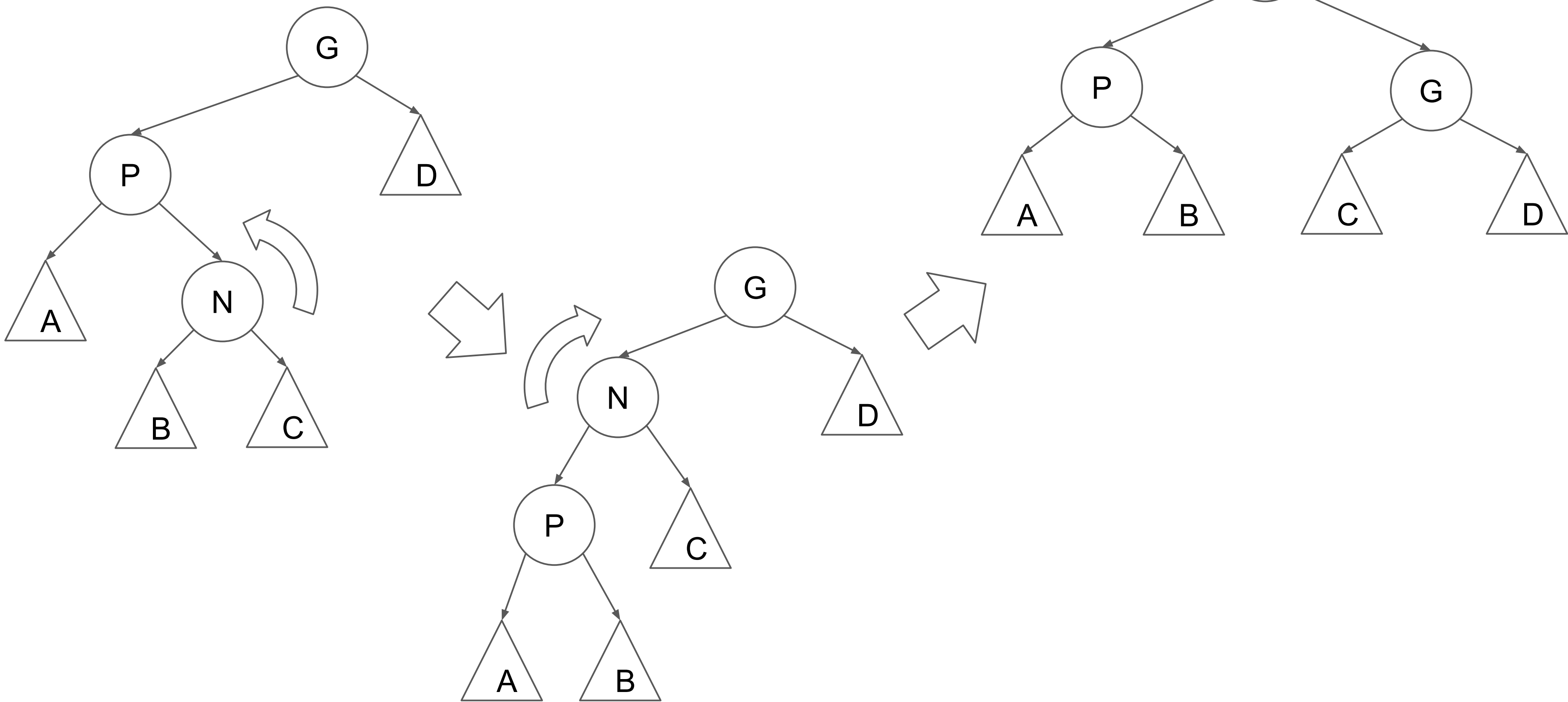
zig-zag (LR)



zig-zag (LR)



zig-zag (LR)

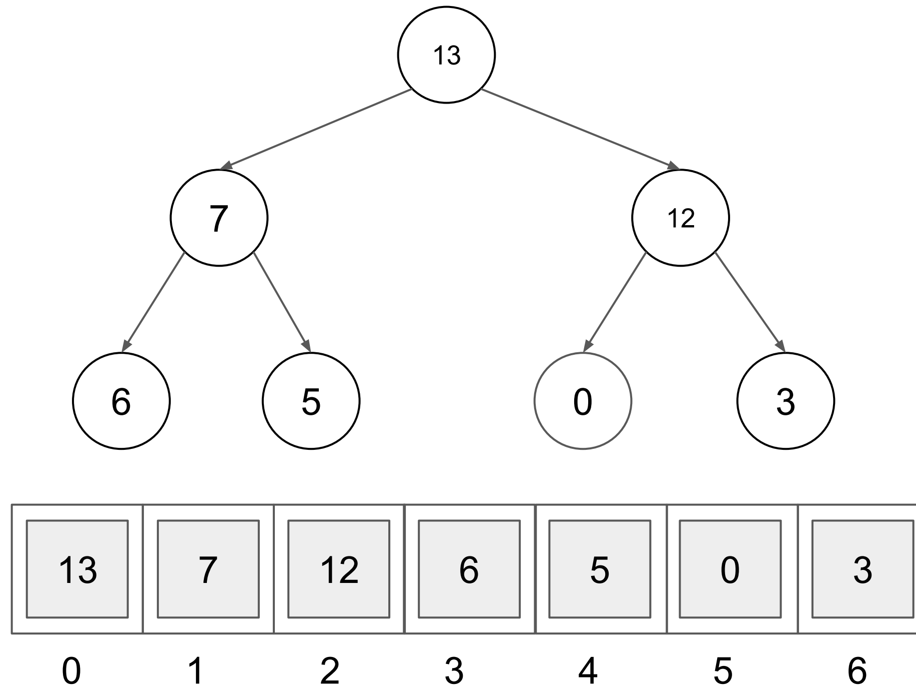


Heaps

Priority queues

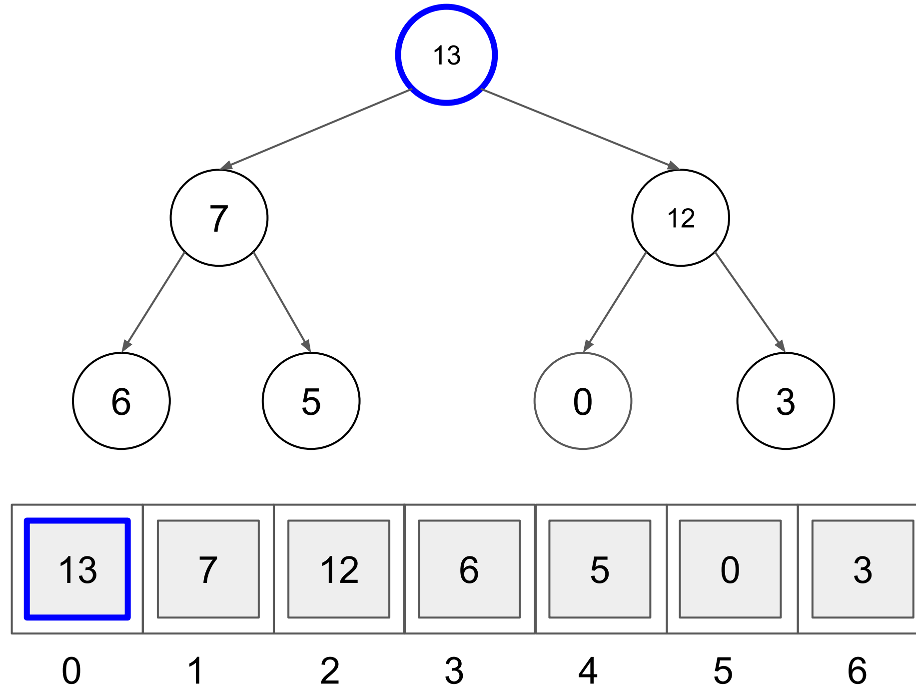
Heapsort

(in-pace version)



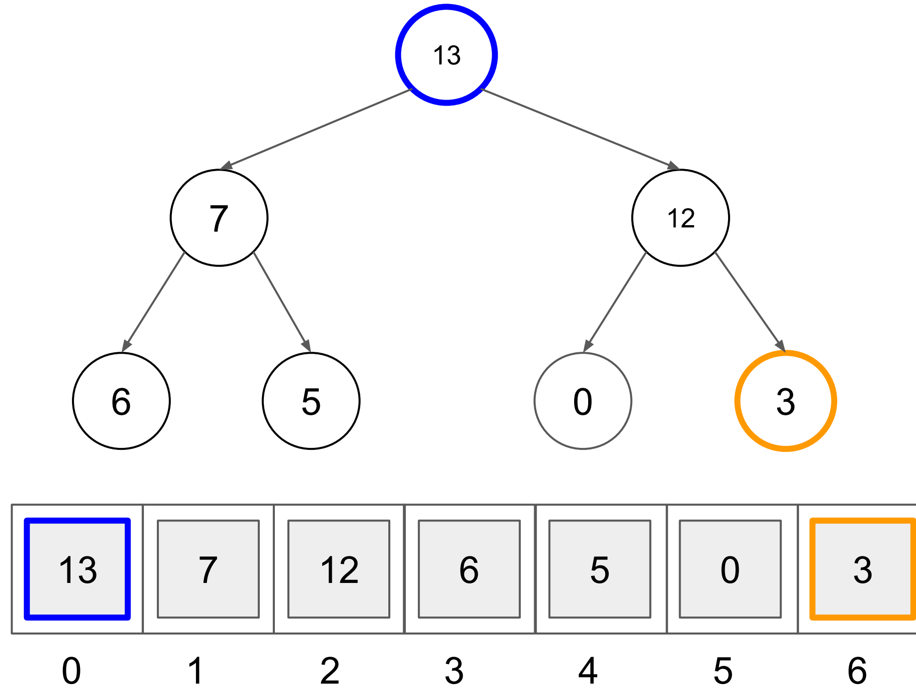
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



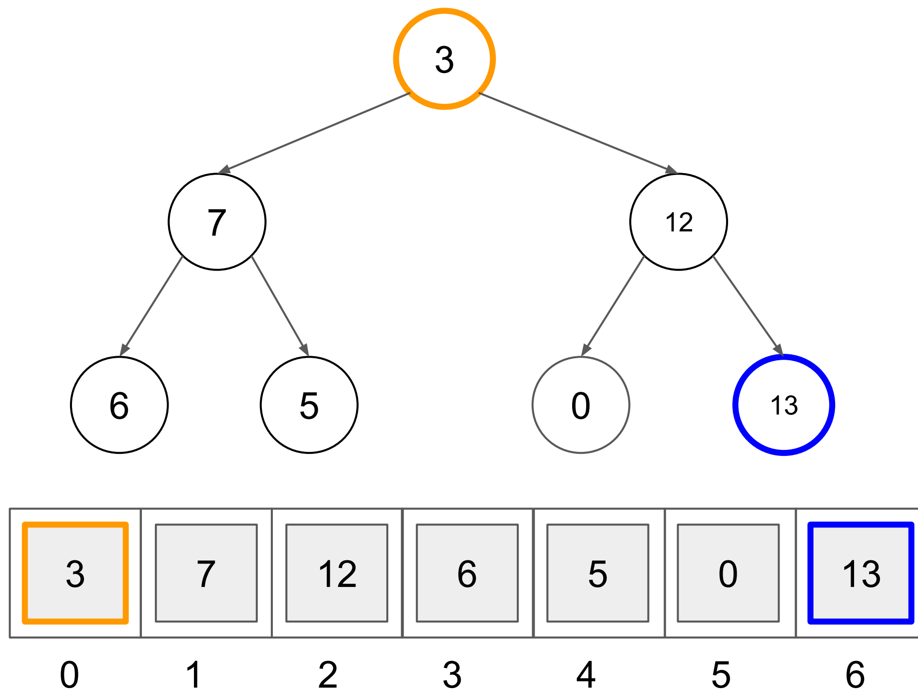
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



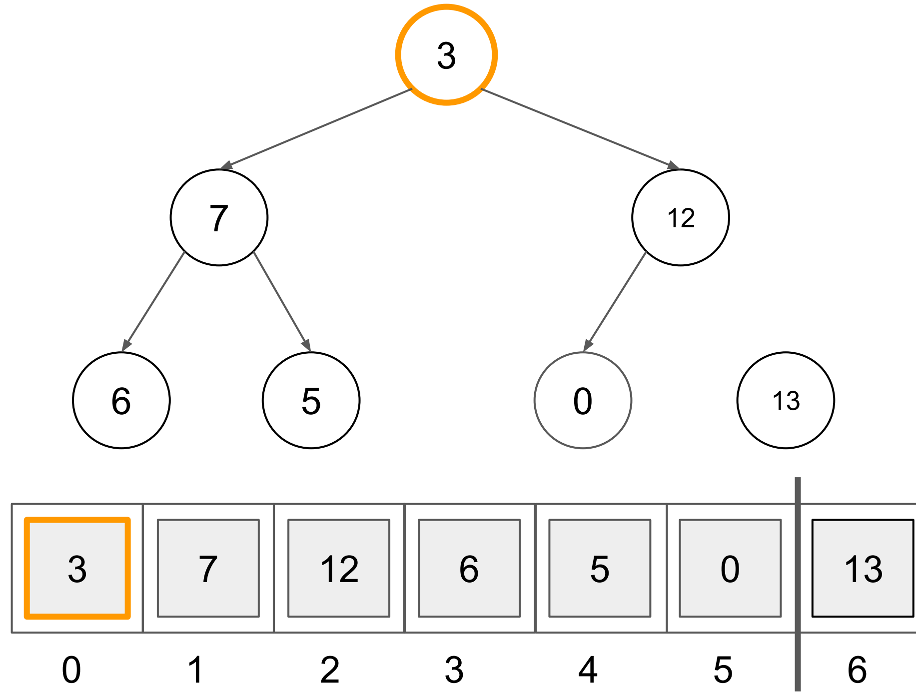
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



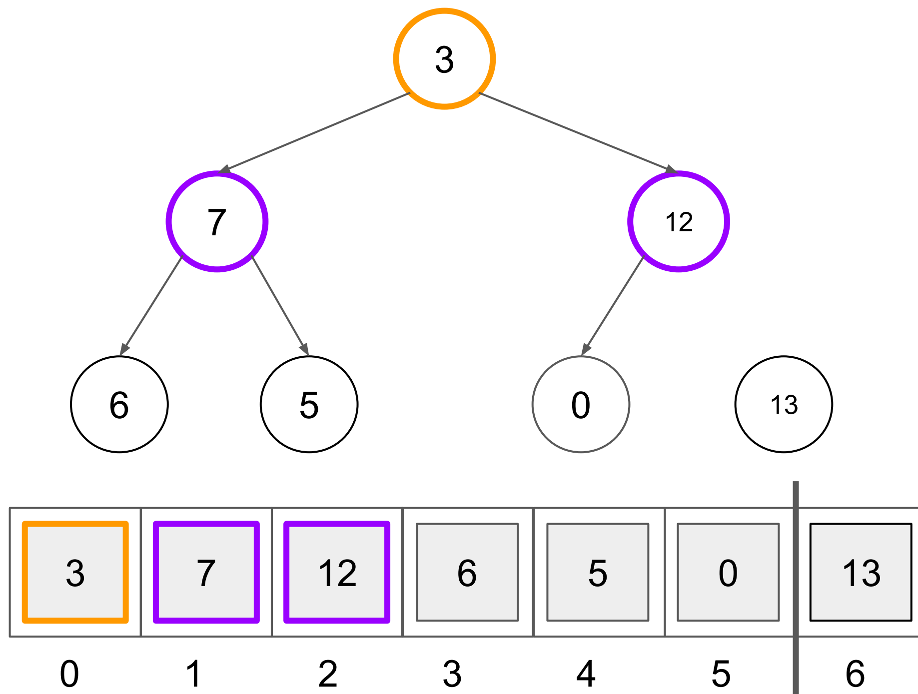
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



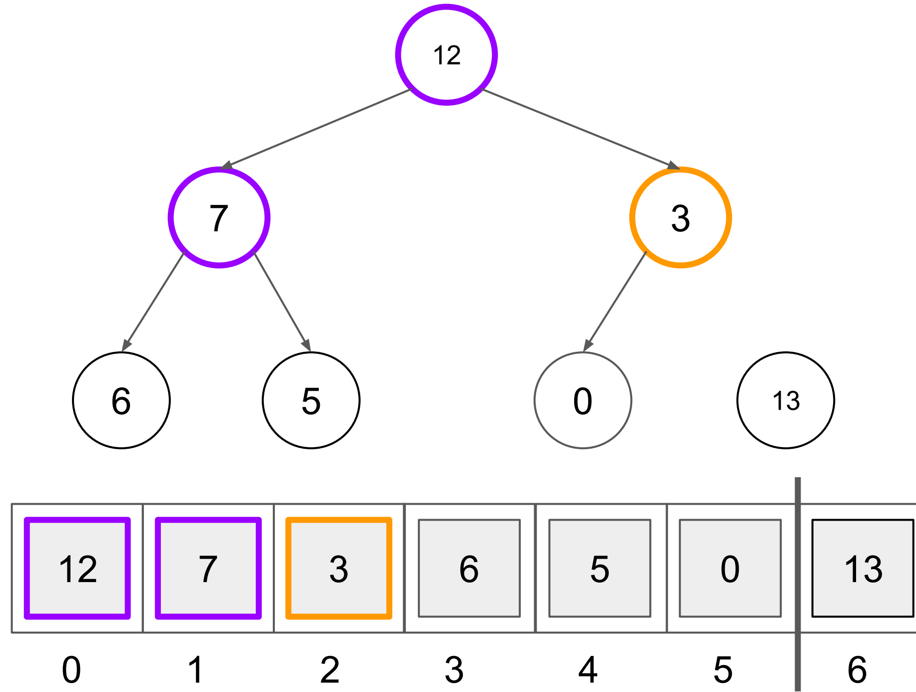
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



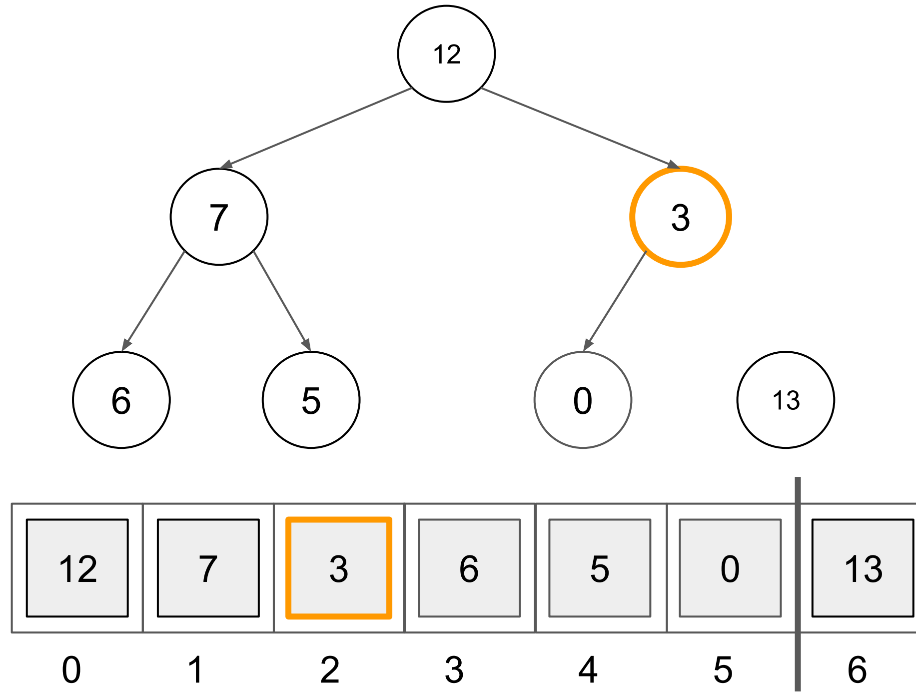
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



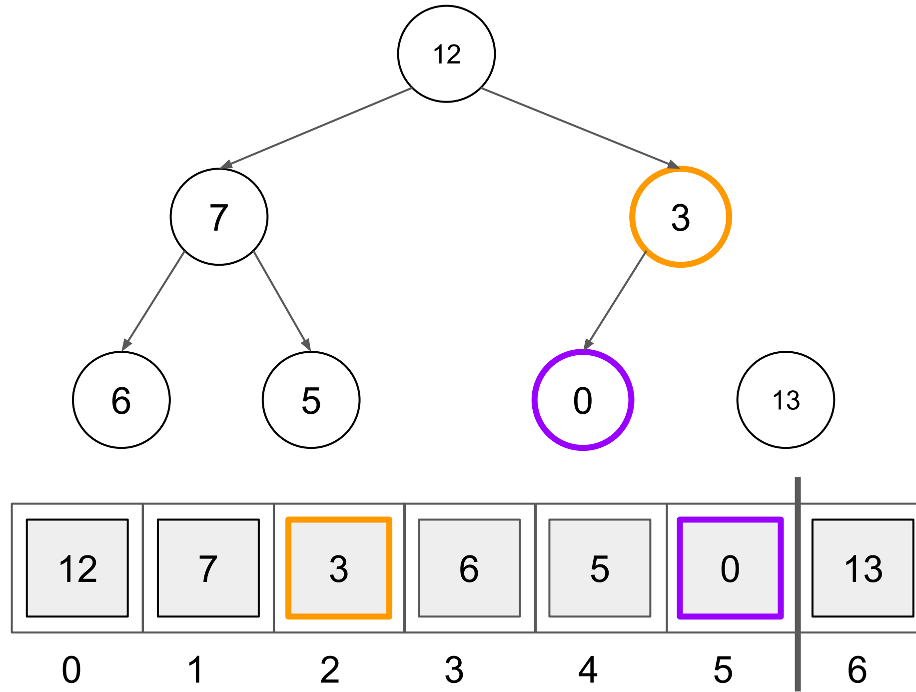
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



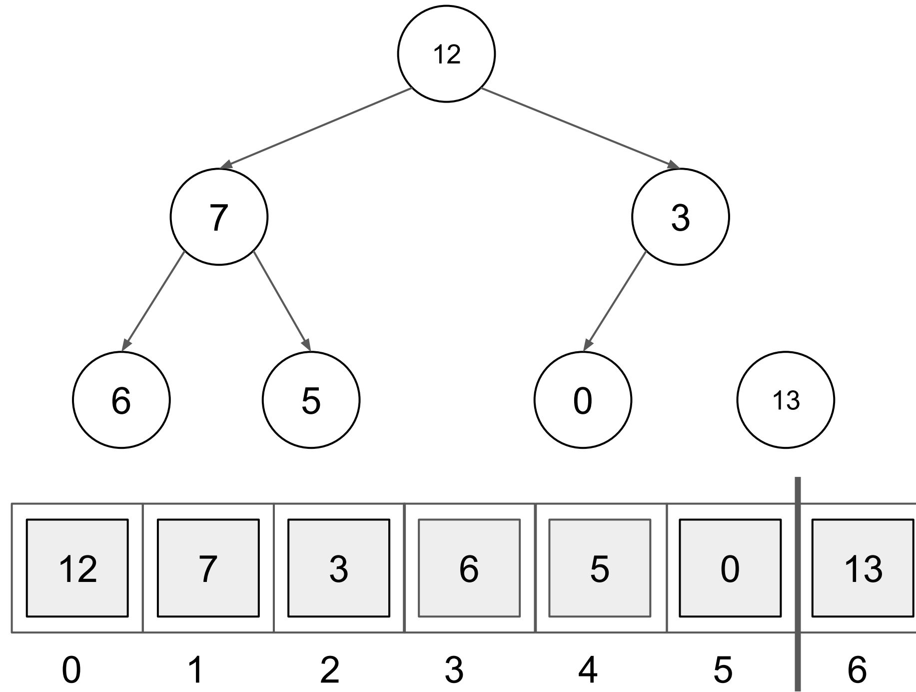
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



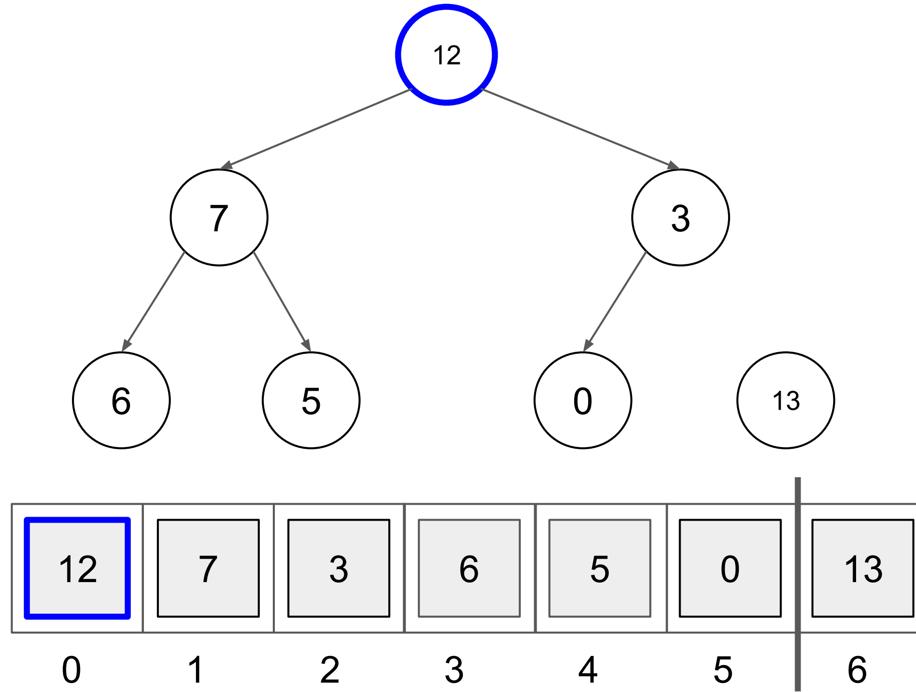
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



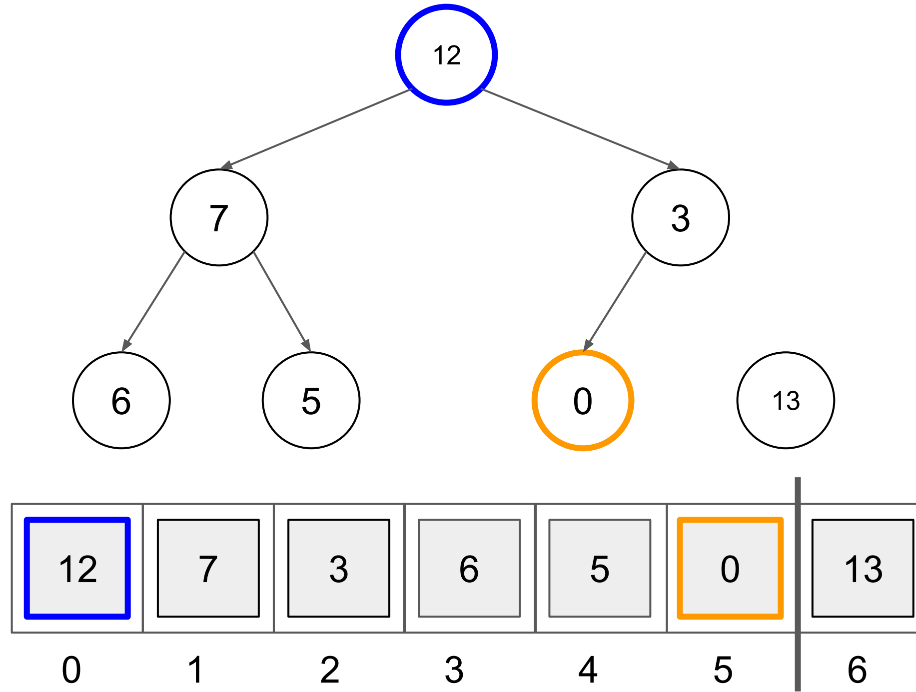
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



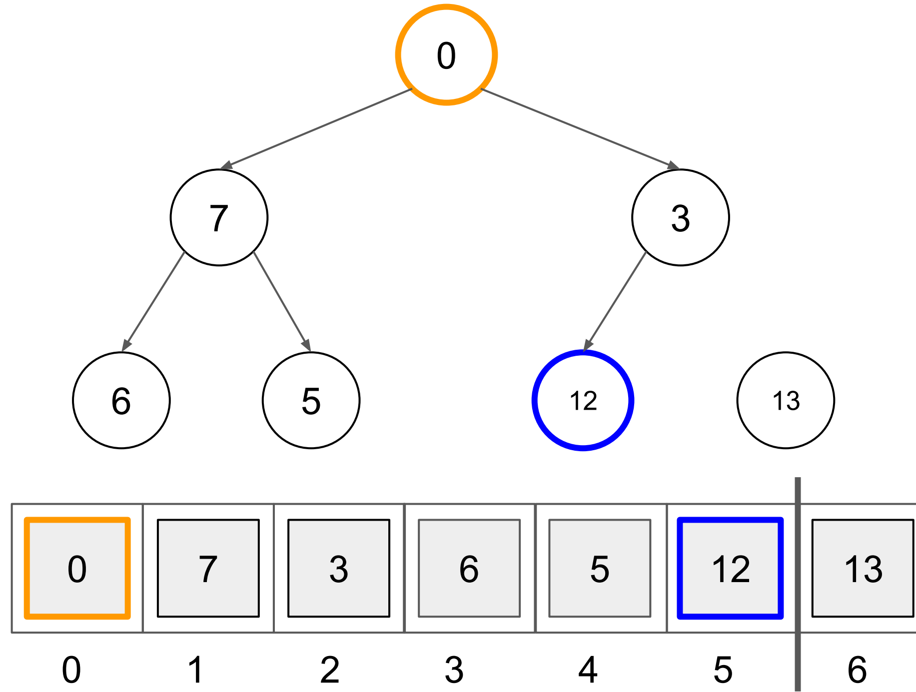
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



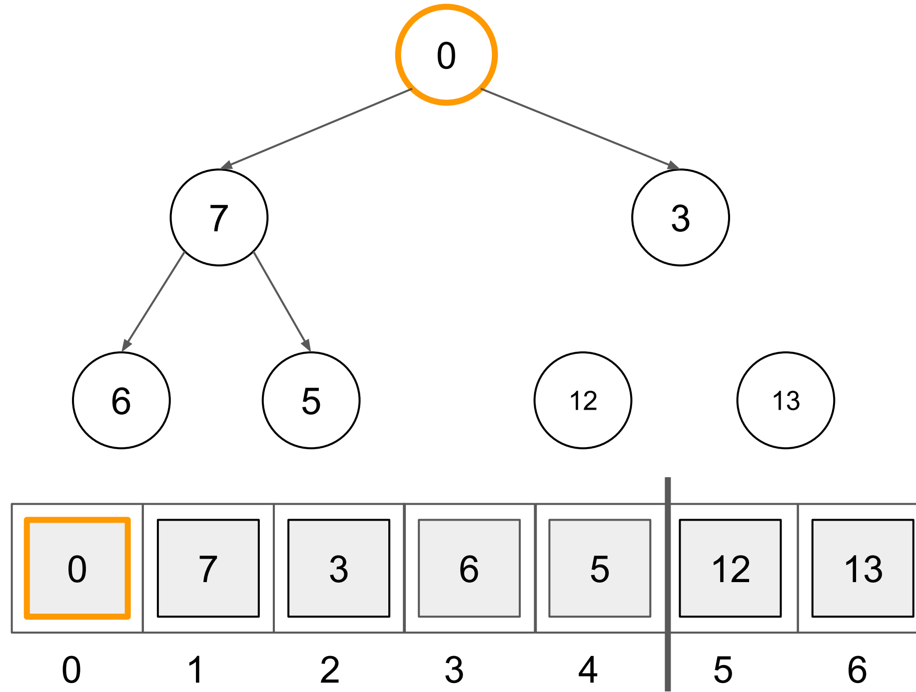
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



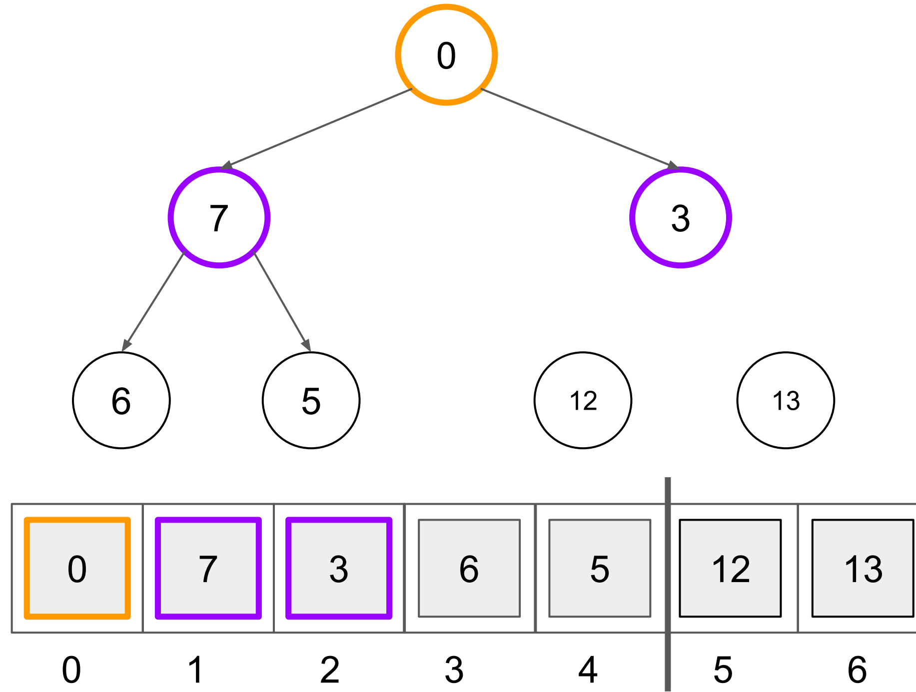
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



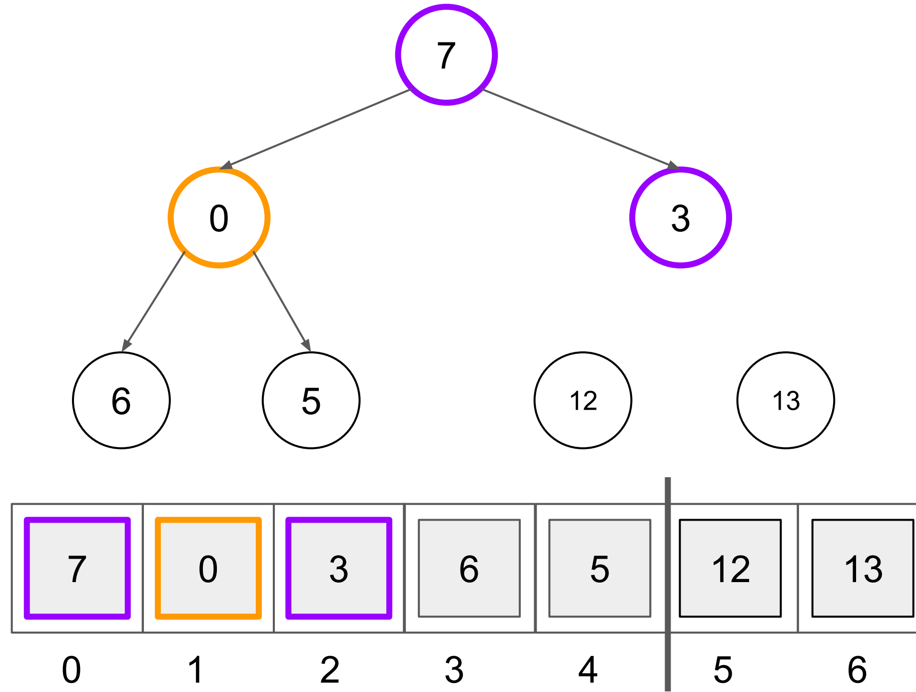
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



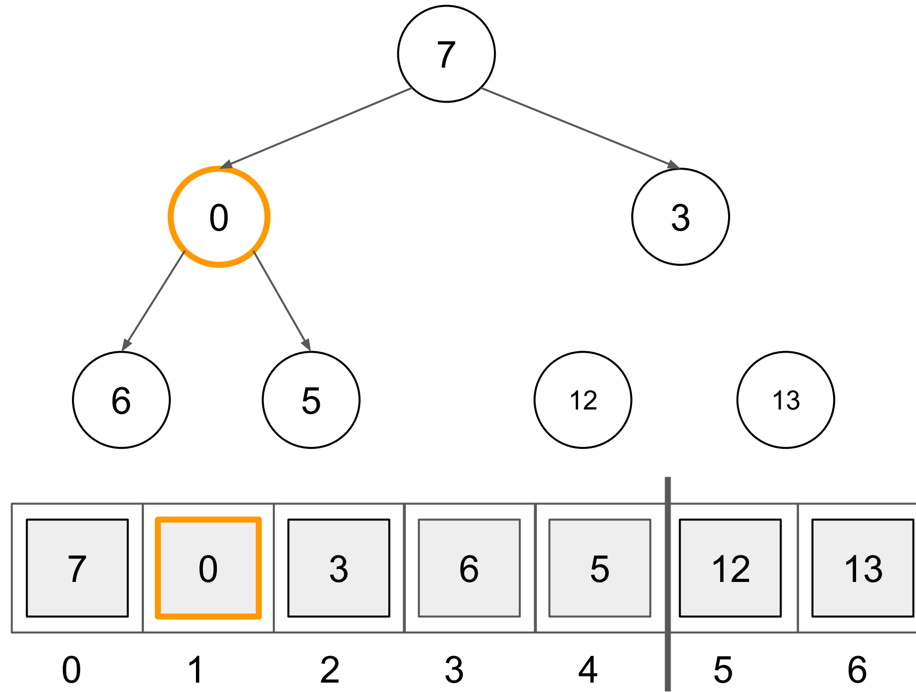
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



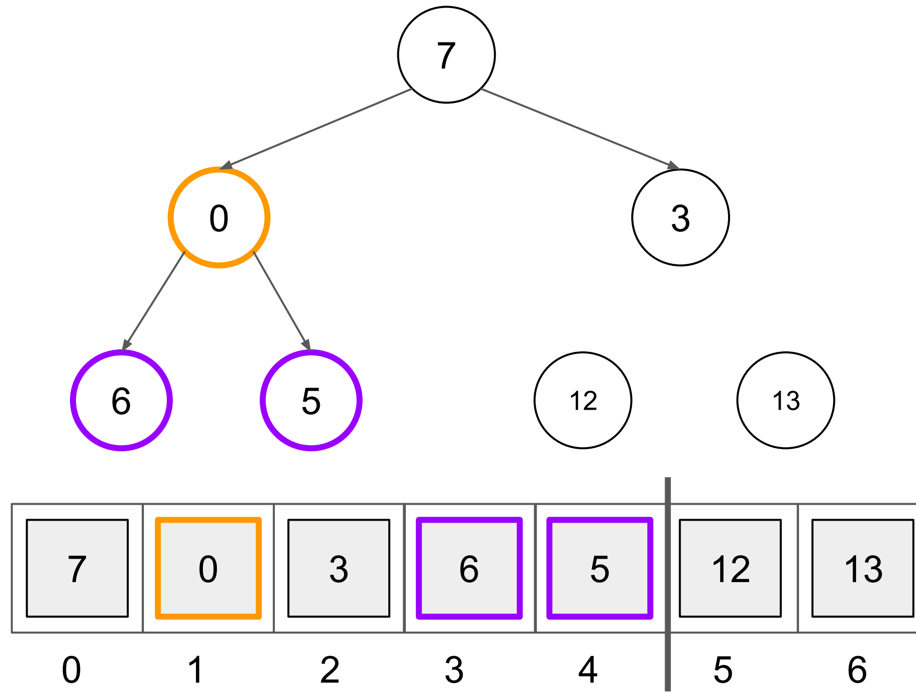
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



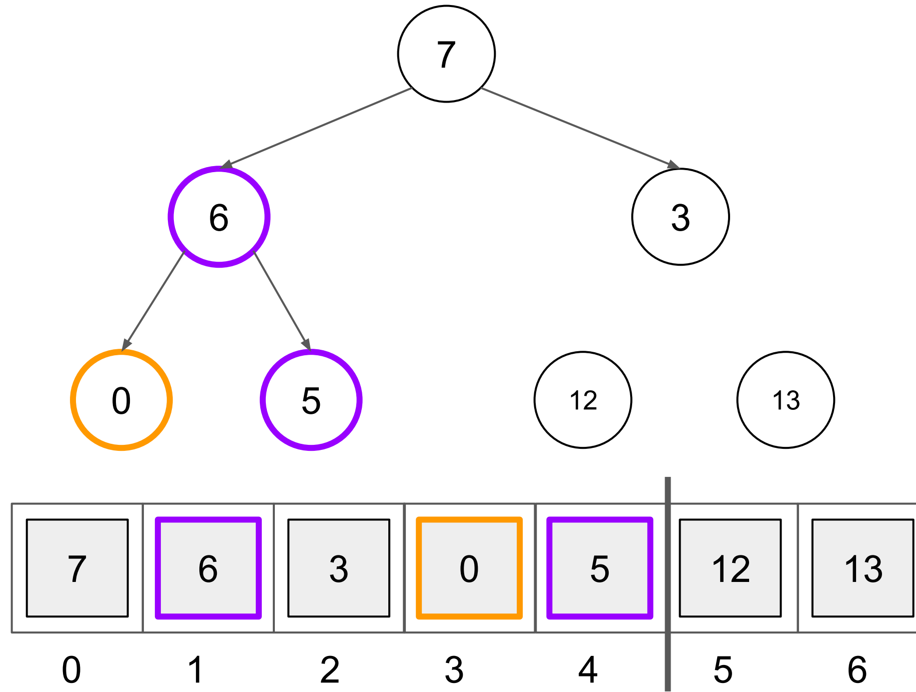
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



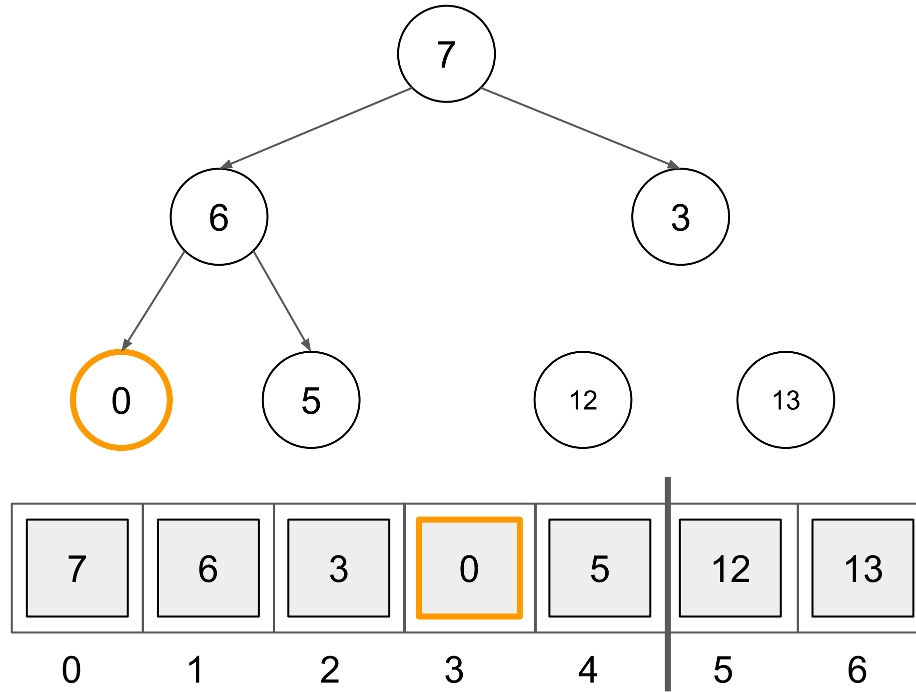
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



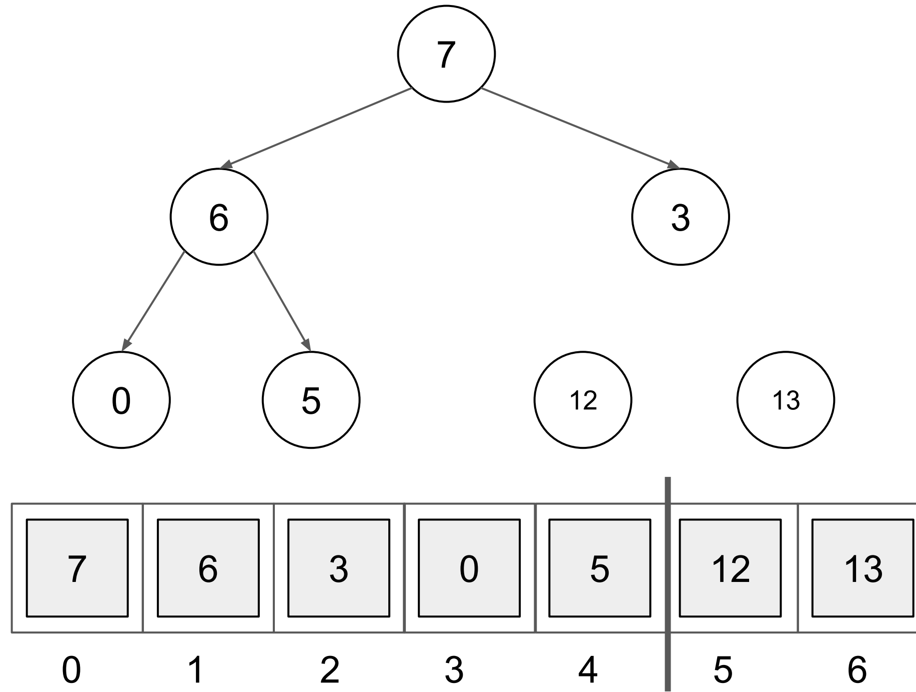
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



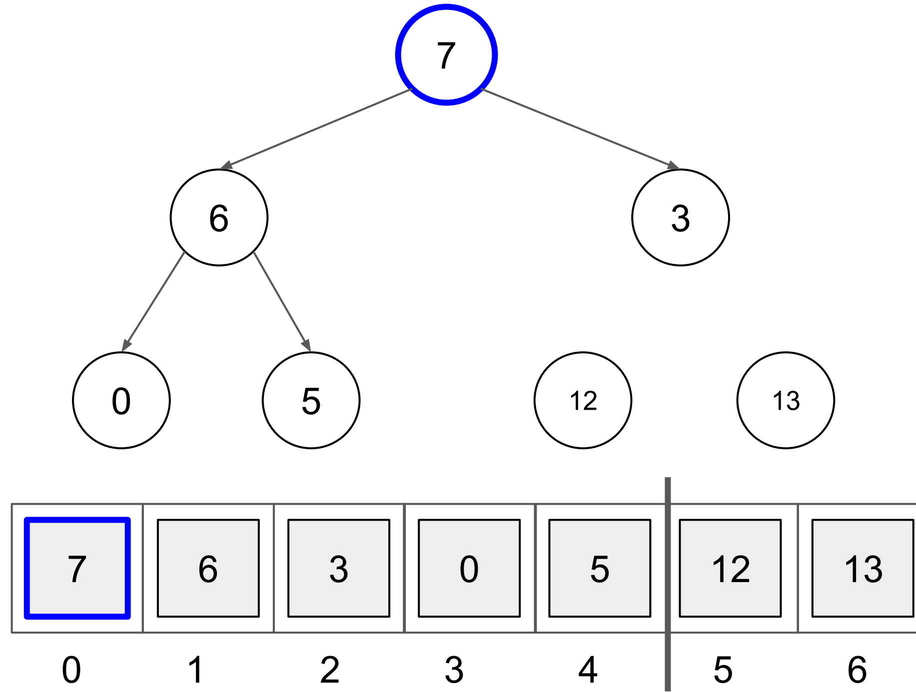
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



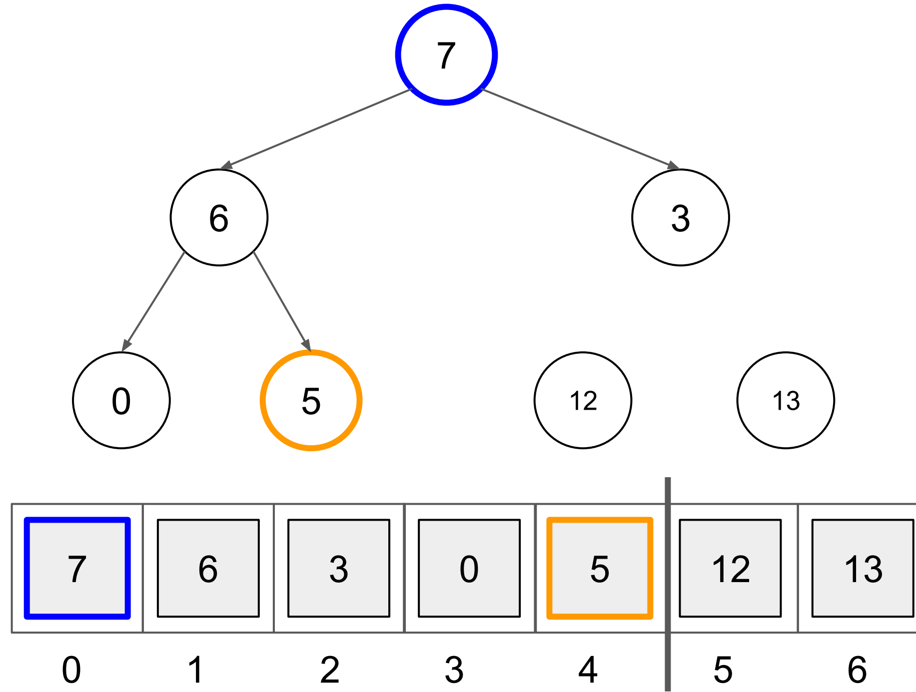
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



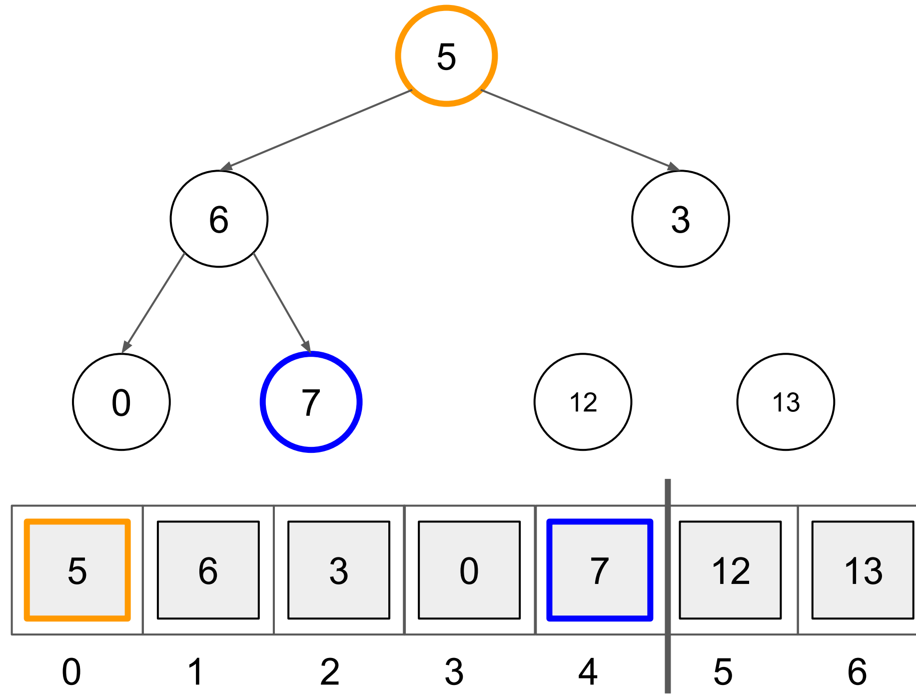
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



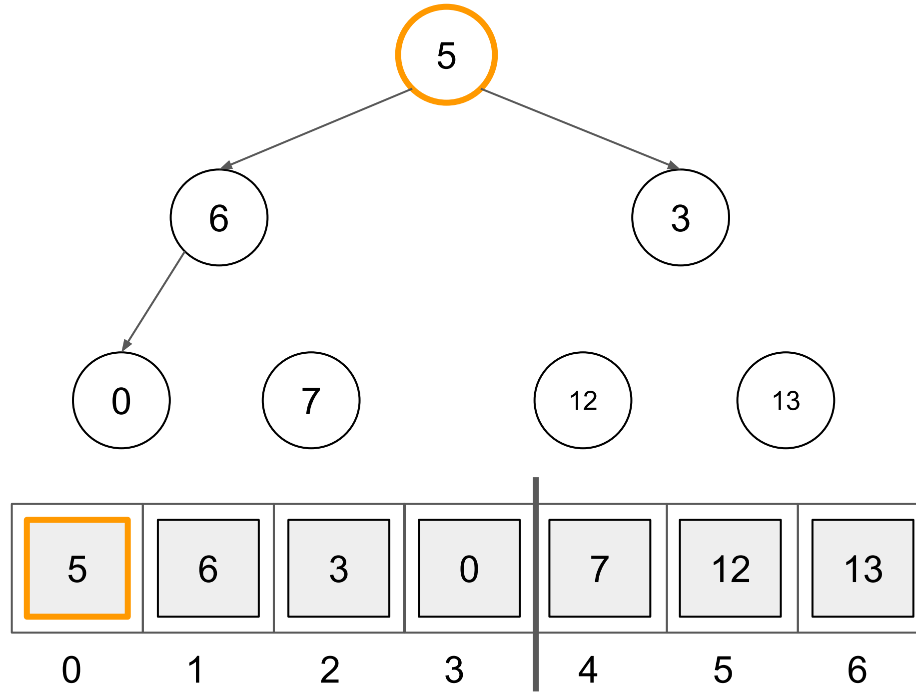
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



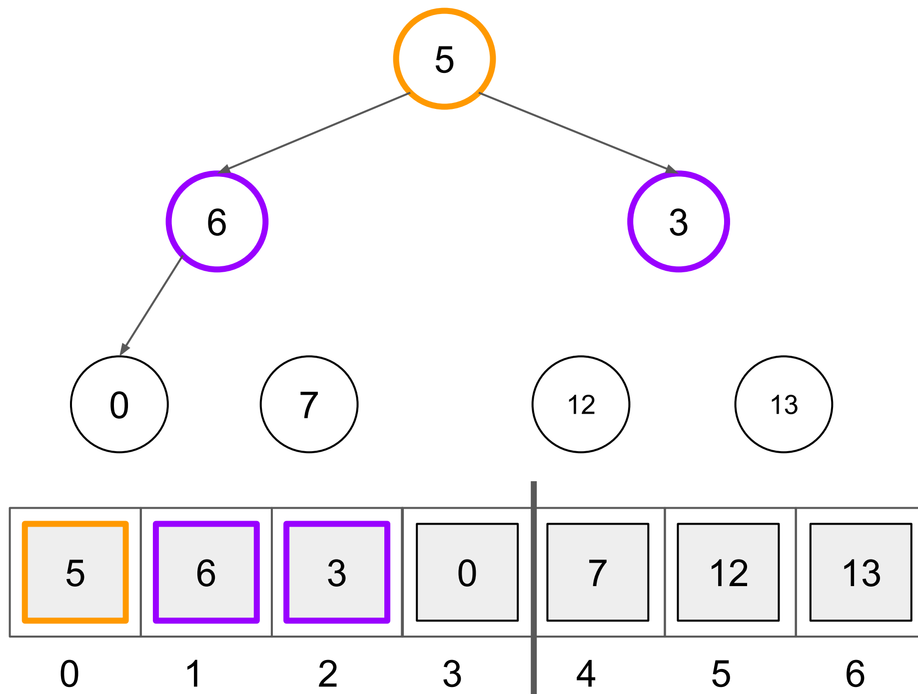
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



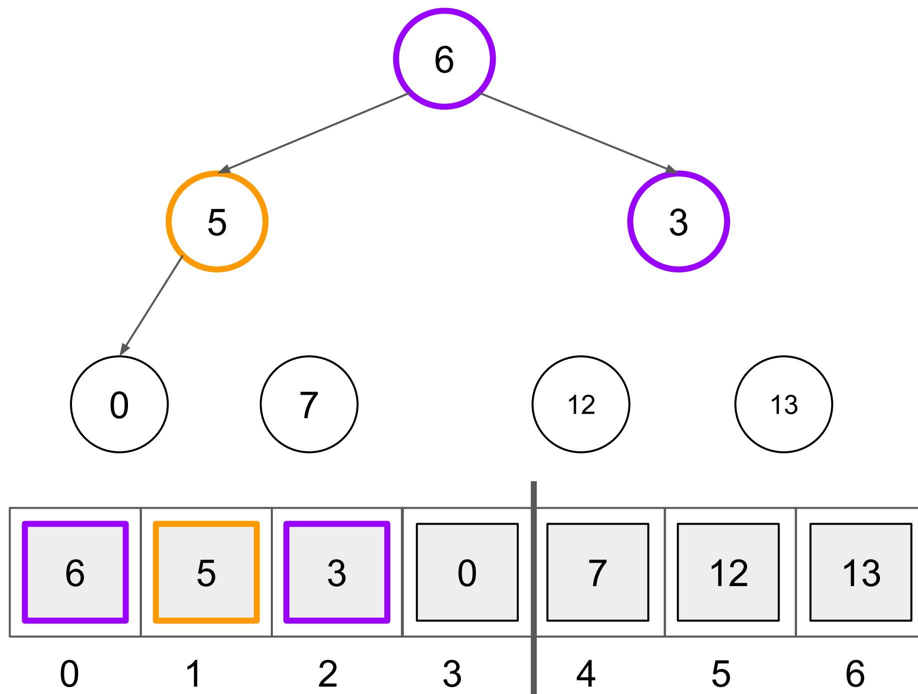
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



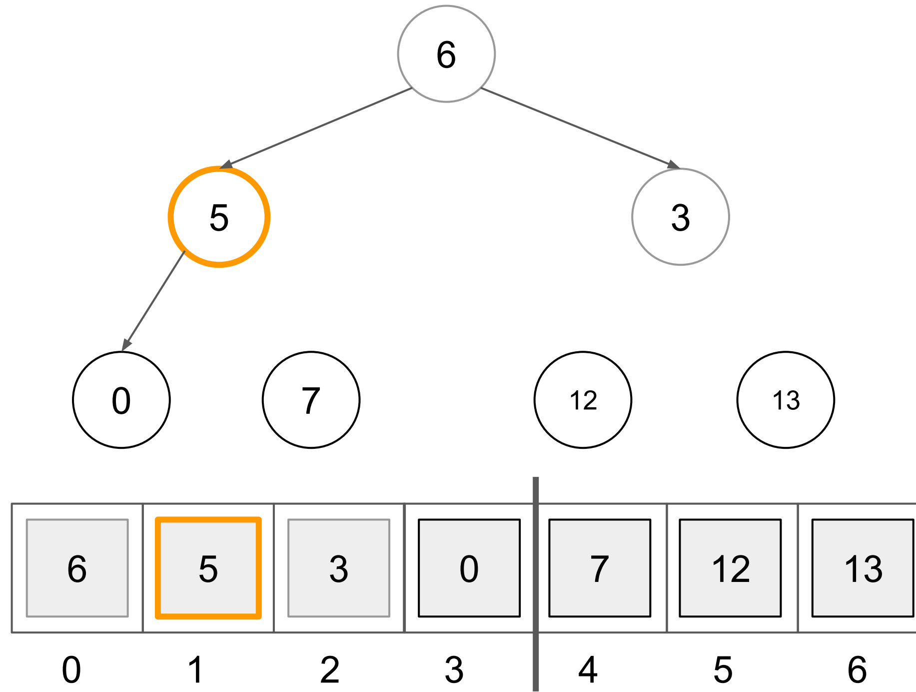
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



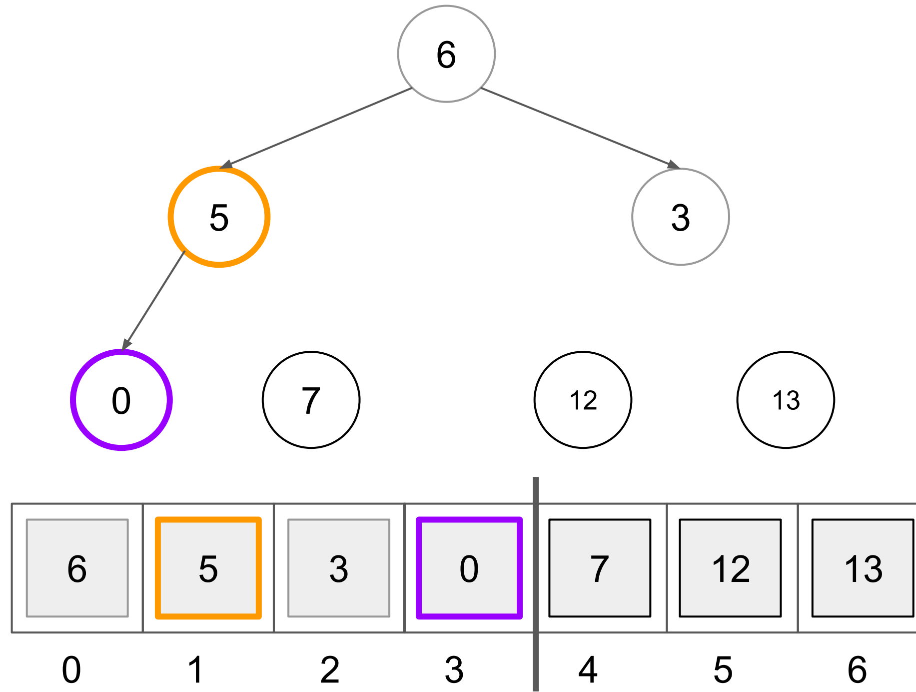
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



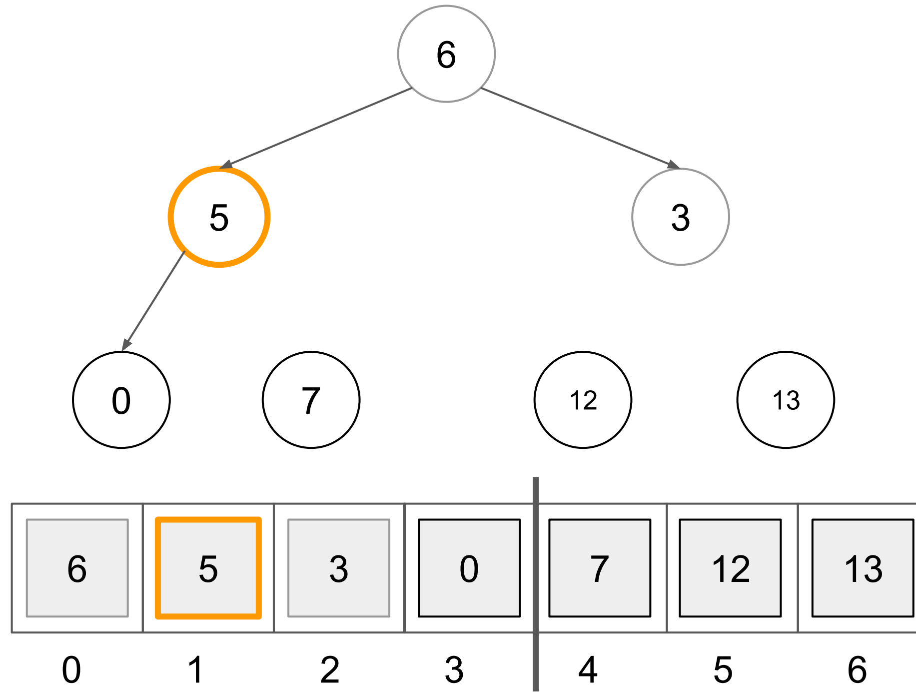
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



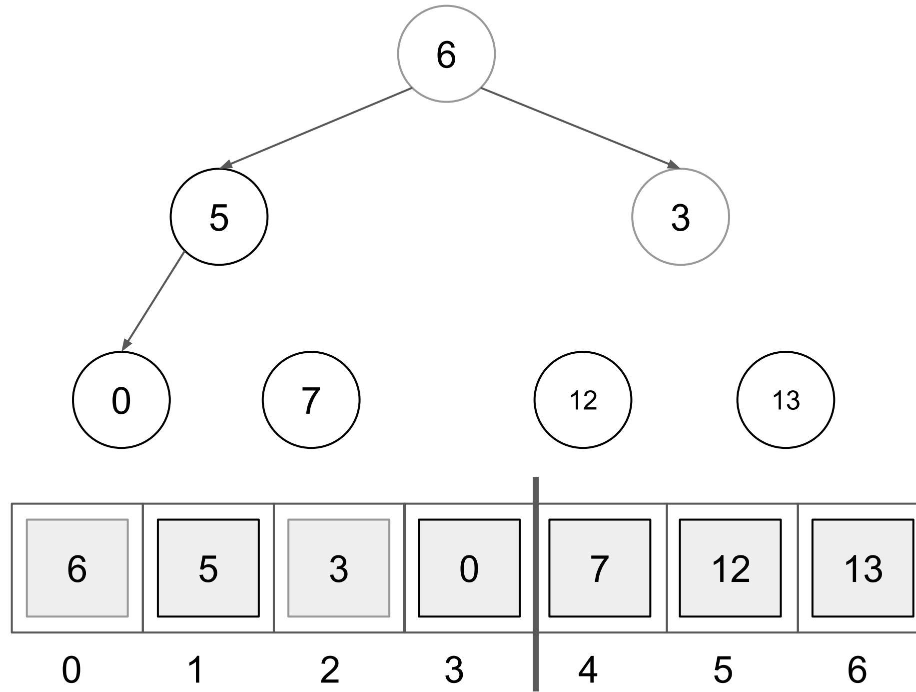
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



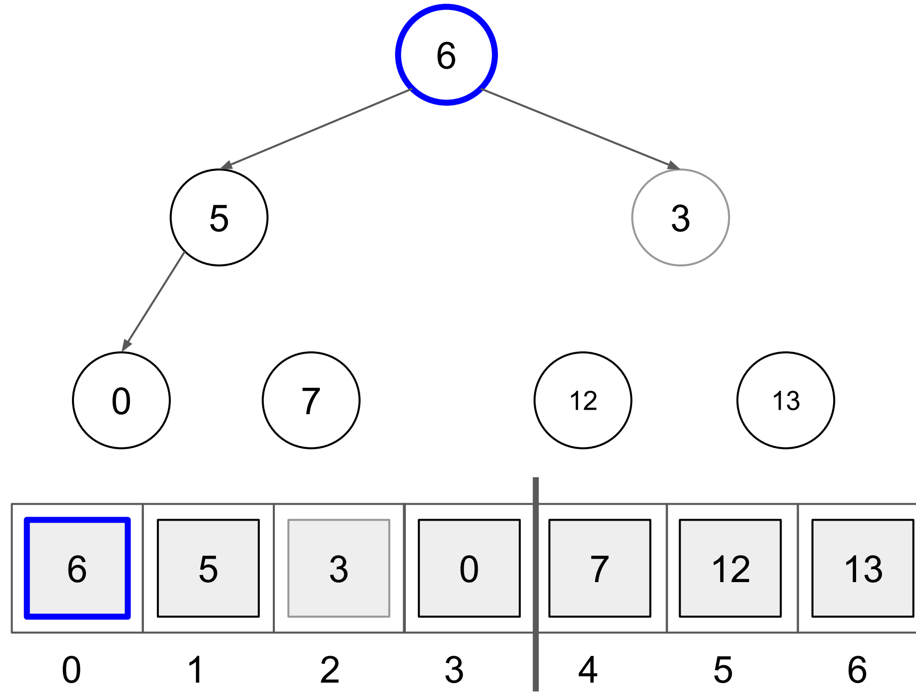
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



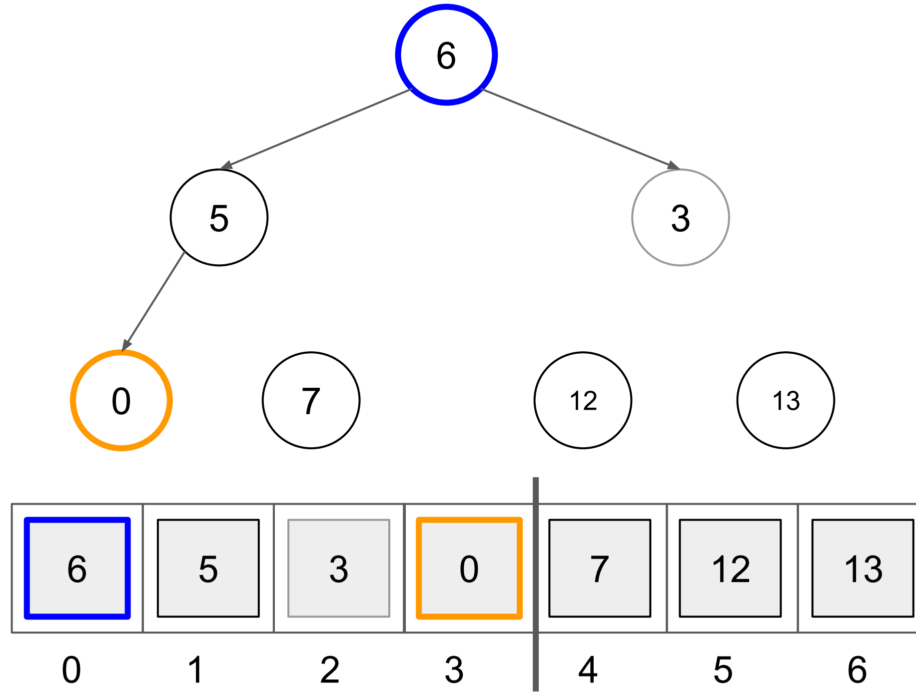
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



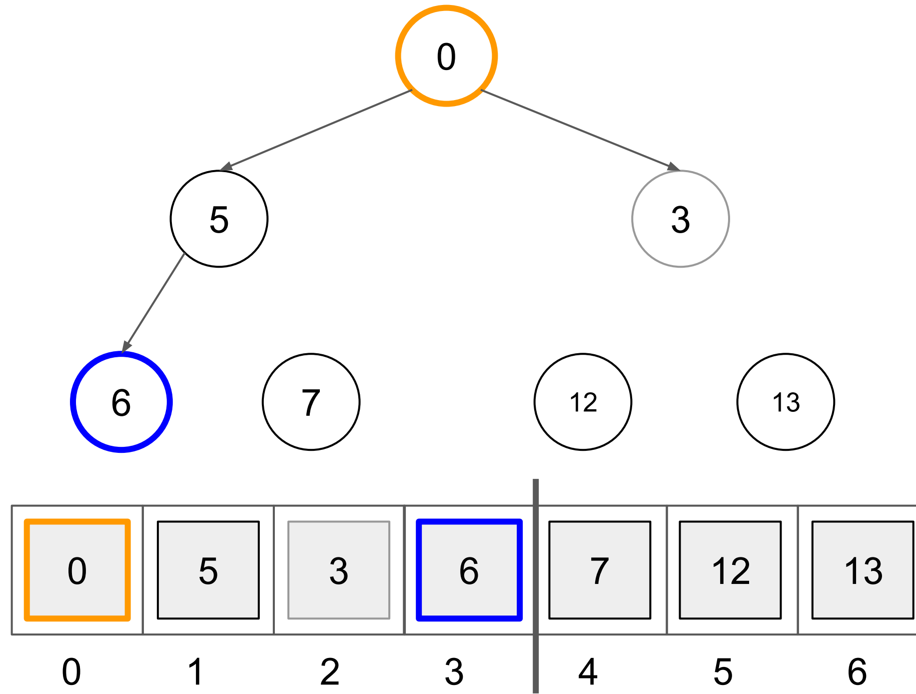
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



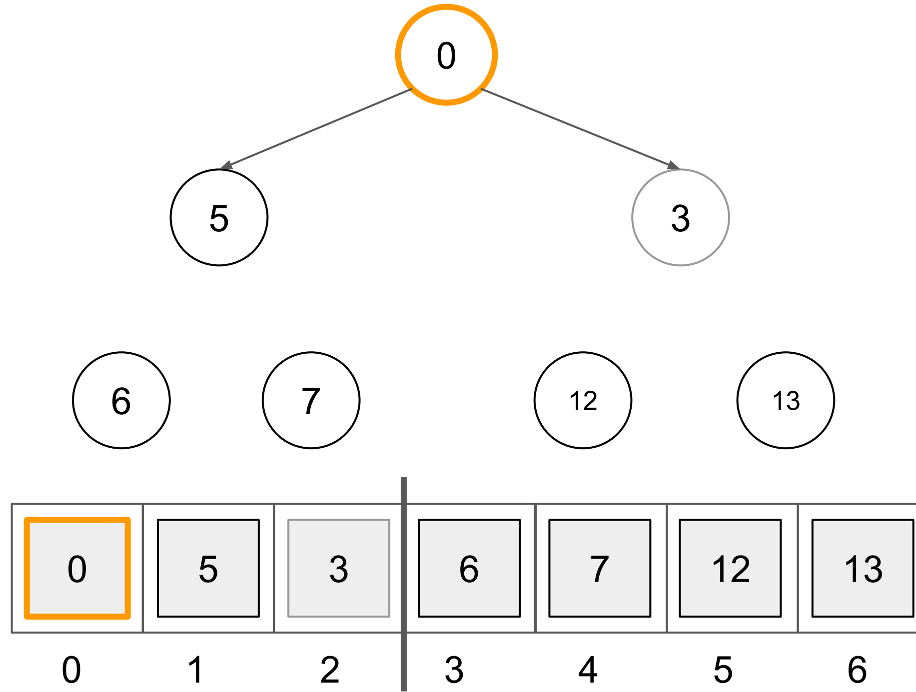
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



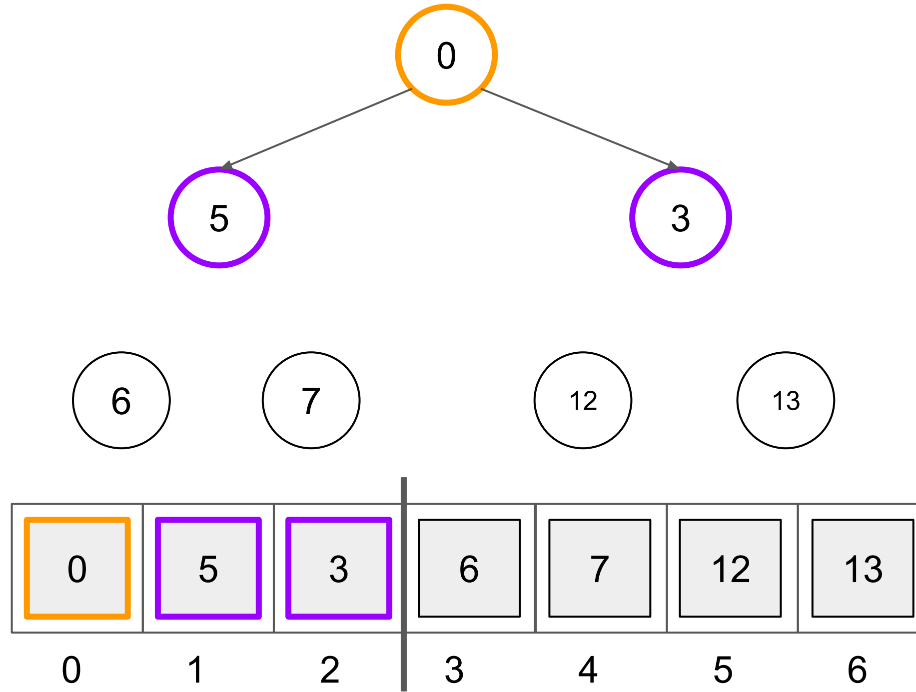
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



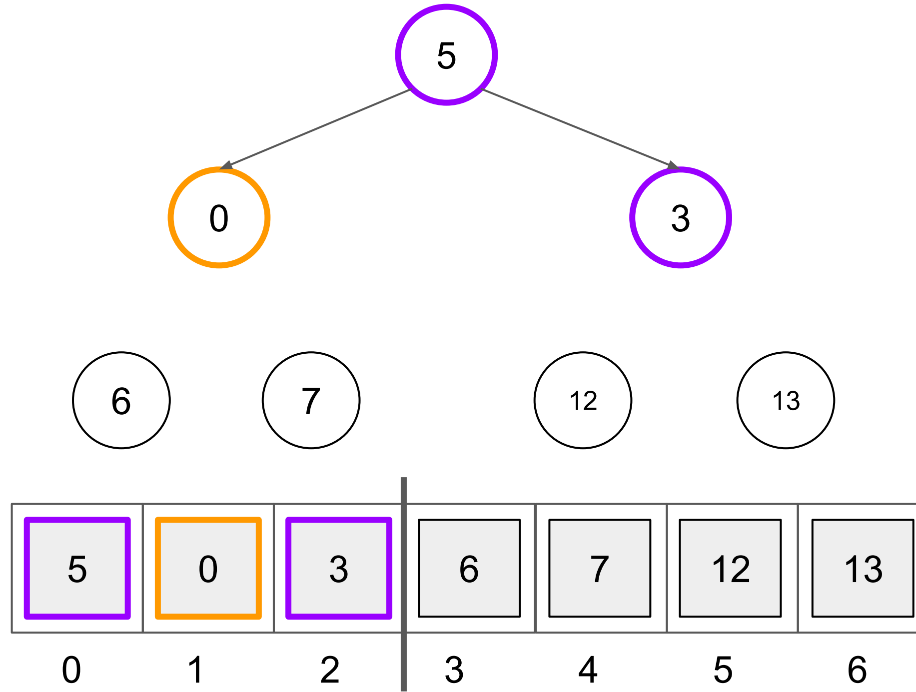
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



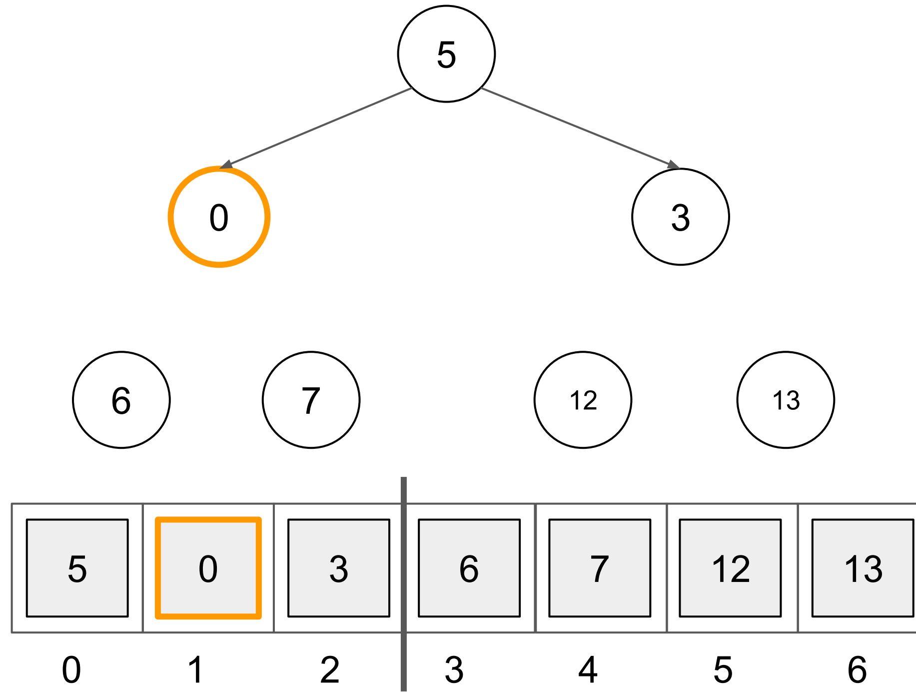
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



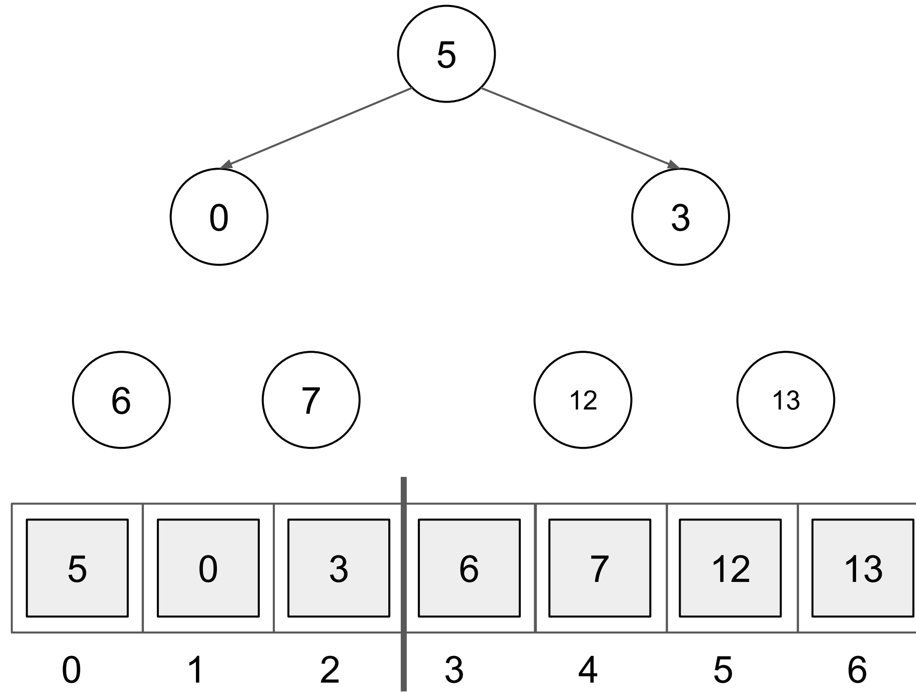
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



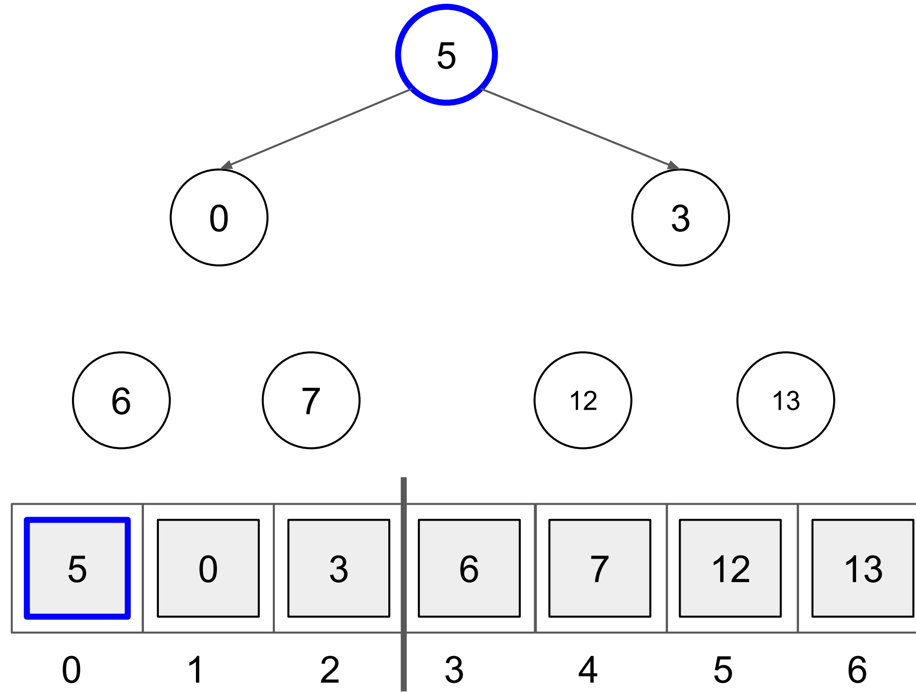
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



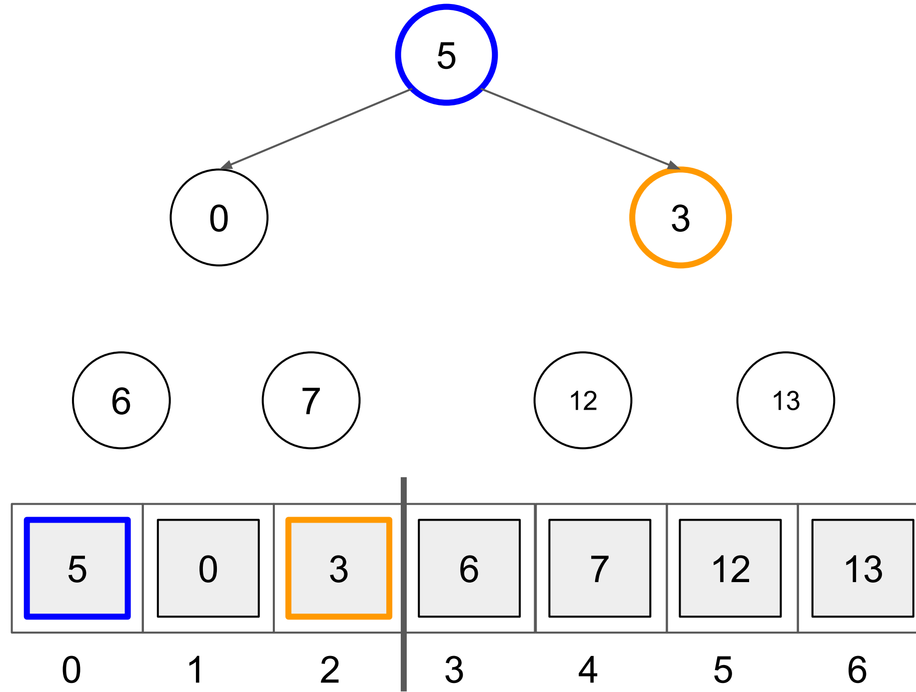
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



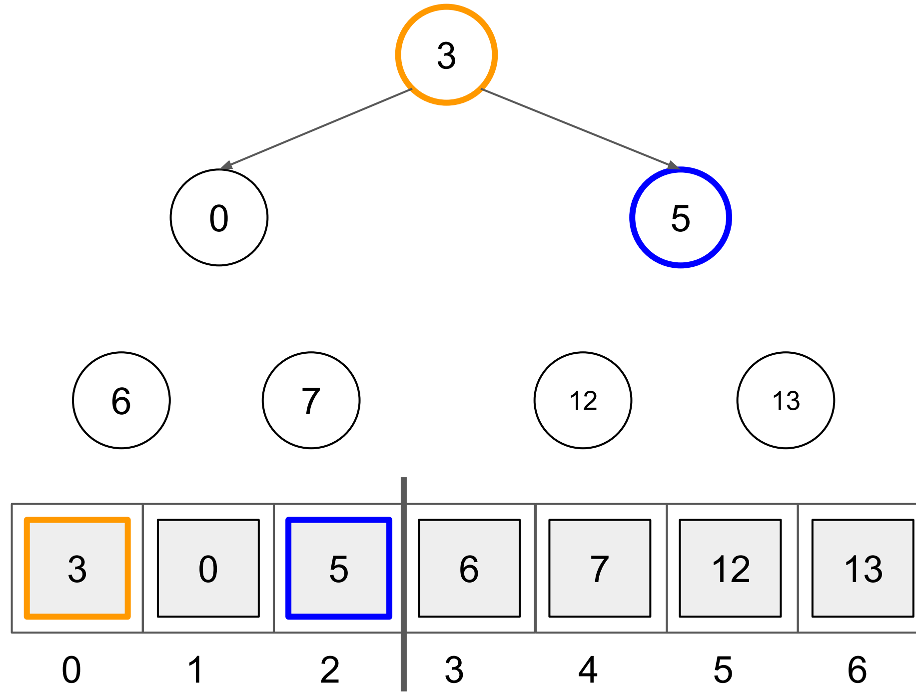
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



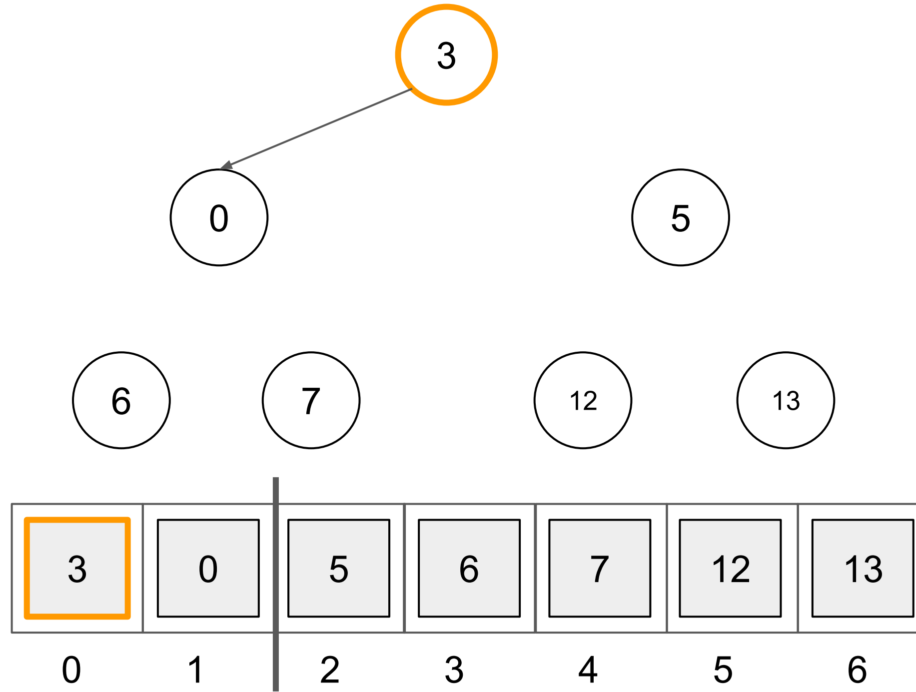
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



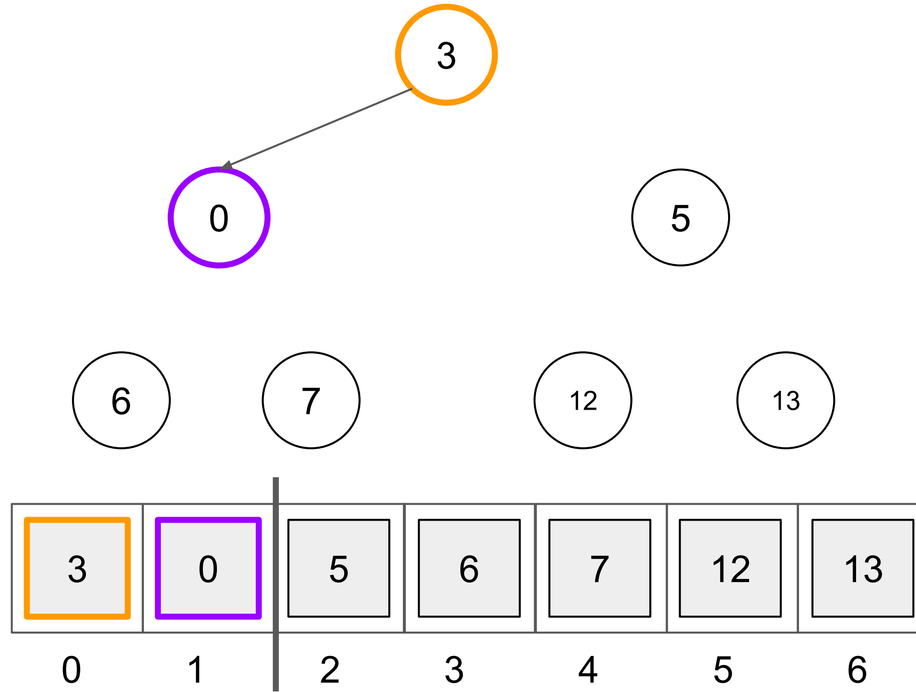
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



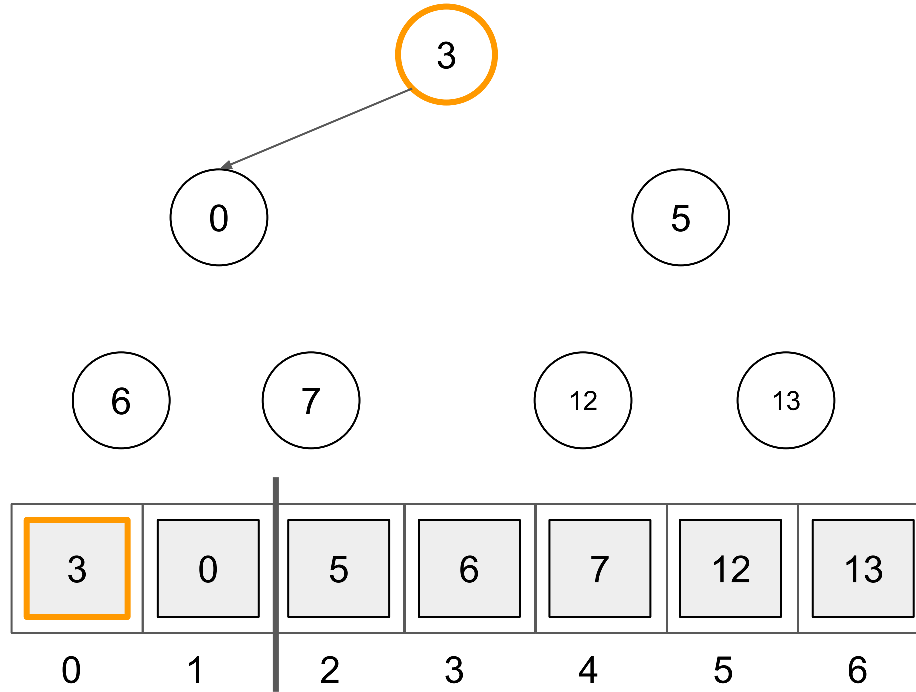
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



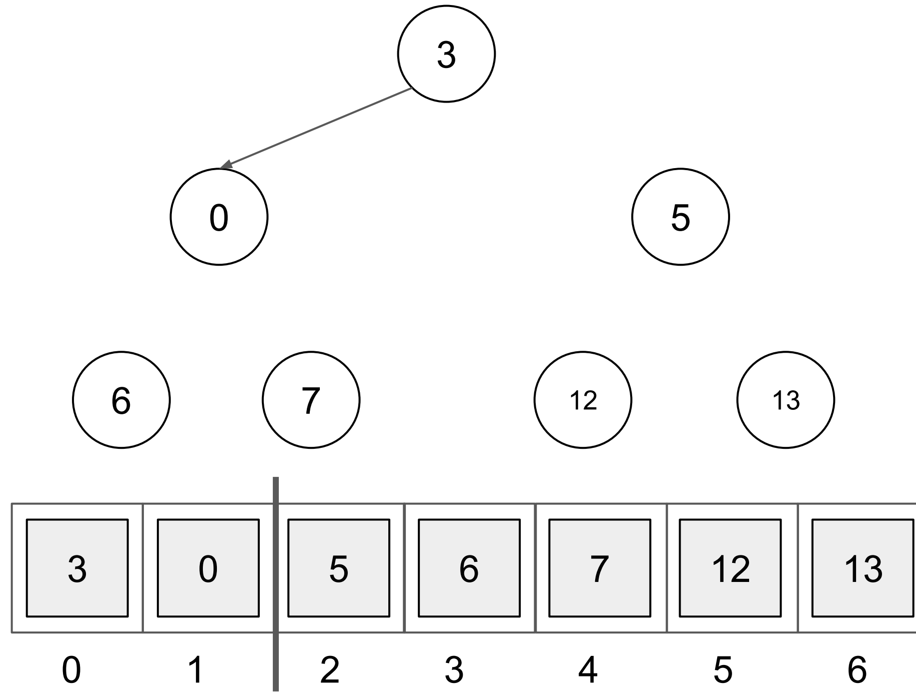
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



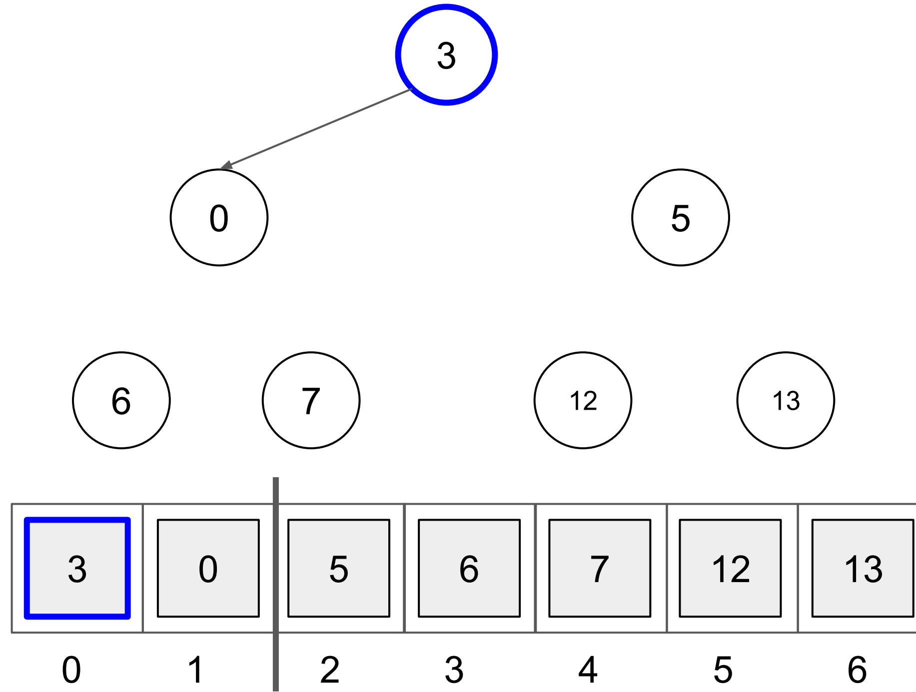
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



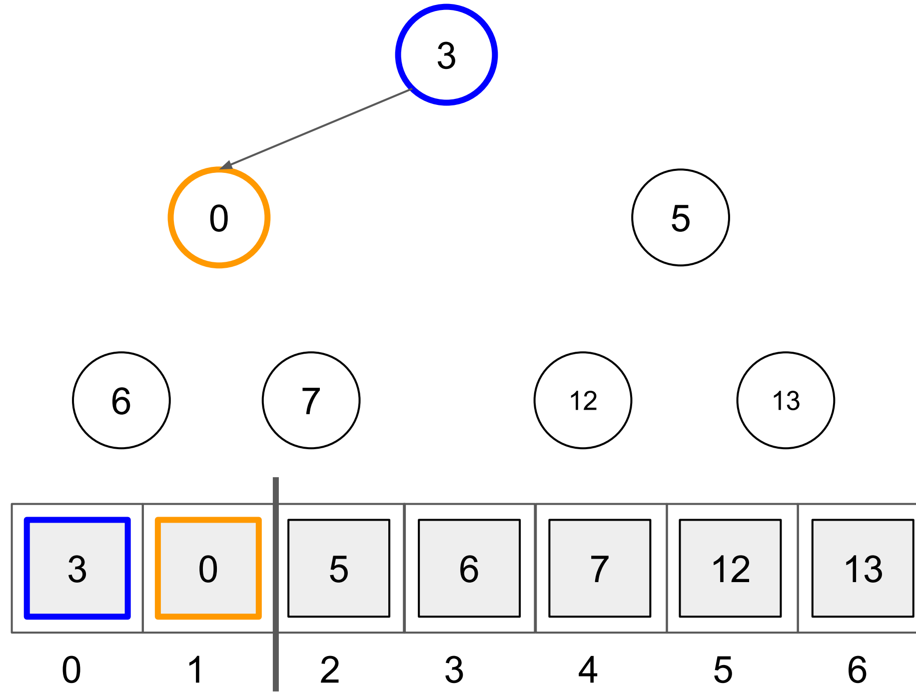
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



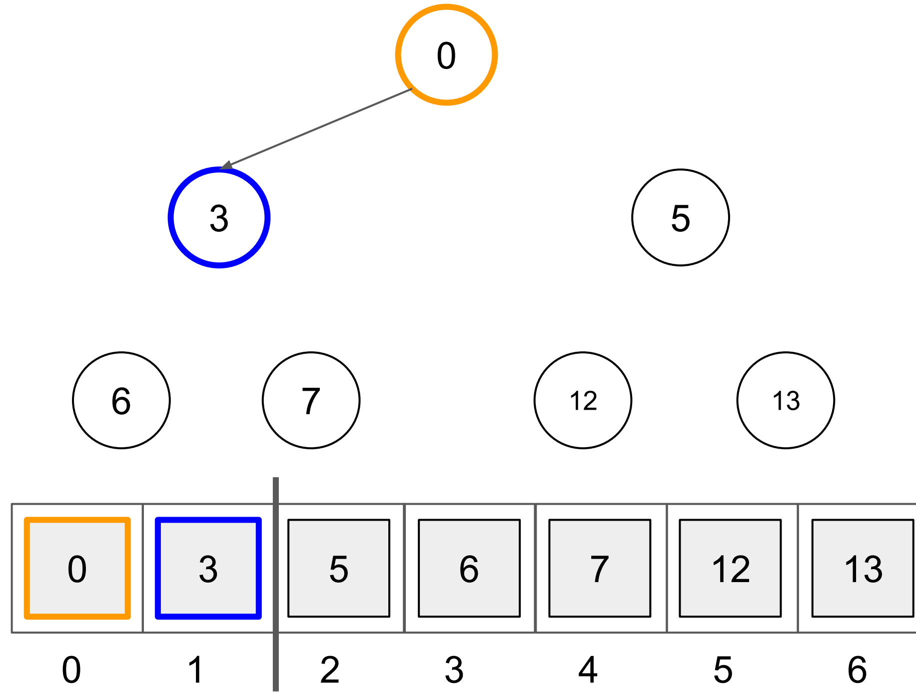
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



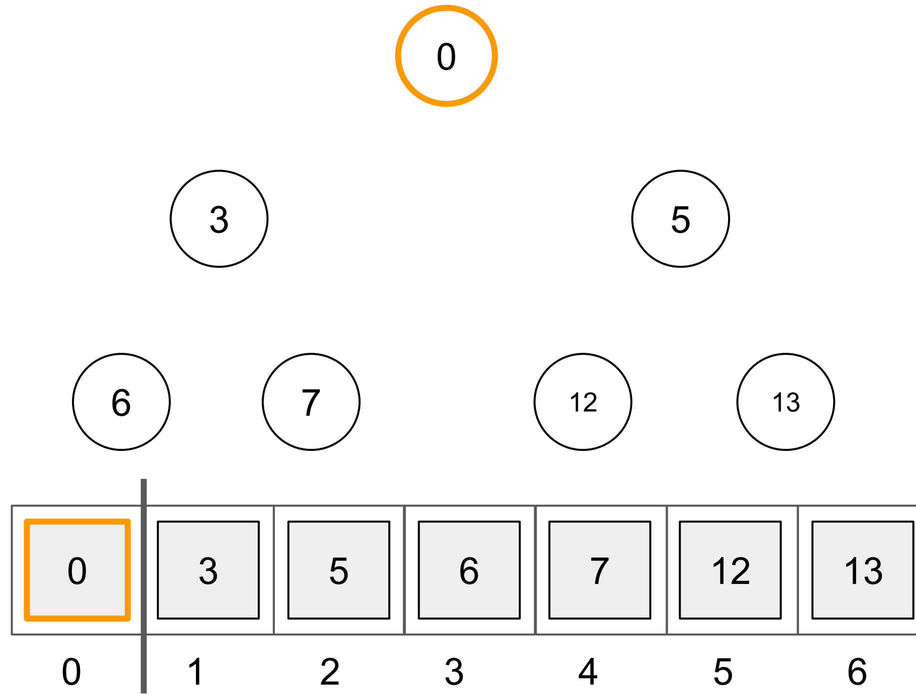
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



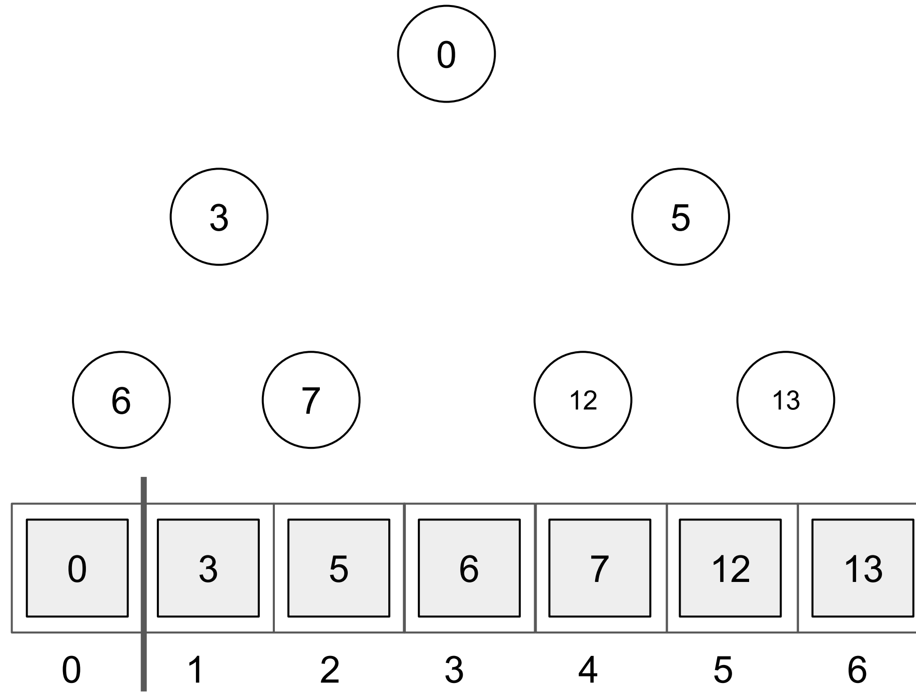
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



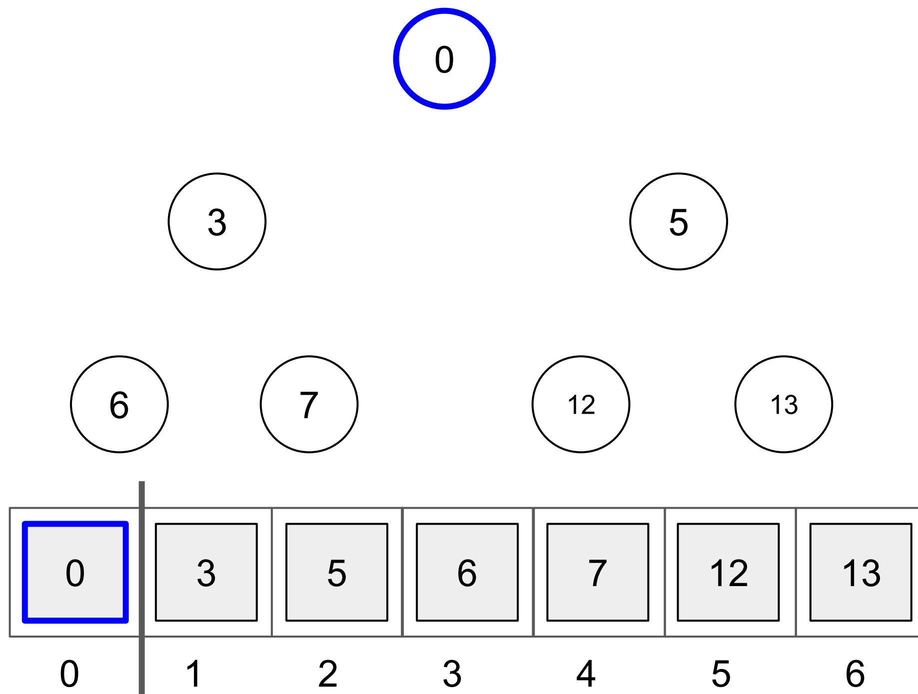
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



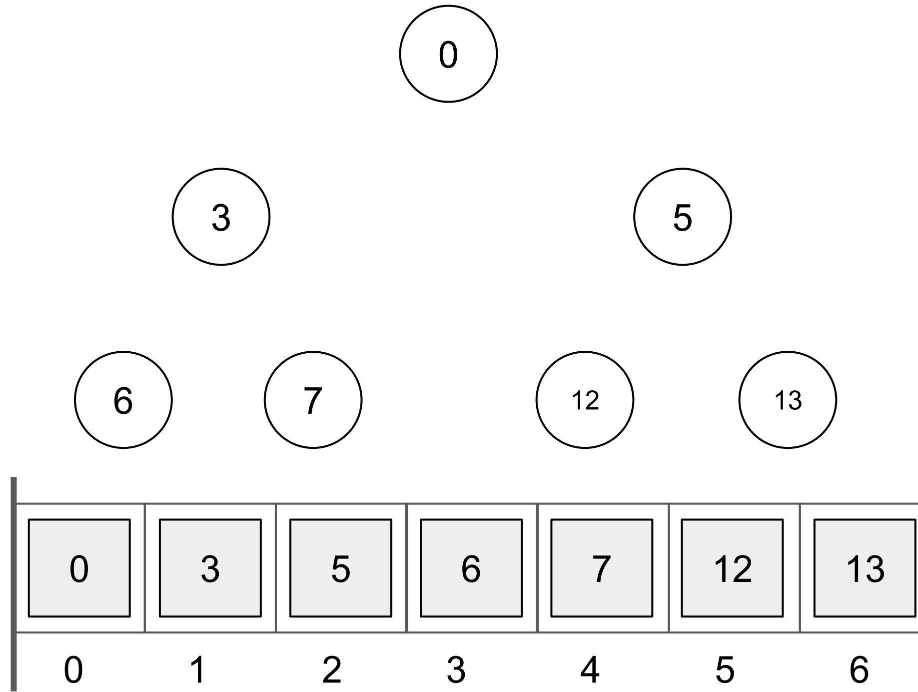
Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c



Parent: p
Children: 2p + 1, 2p + 2

Parent: (c - 1) / 2
Child: c

heap size

The asymptotic running time of heapsort?

Heapsort

Worst-case: $O(n * log(n))$

In-Class Activity

Binary Search

The asymptotic running time of
binary search performed on a sorted array?

(Notes from the live demo or live coding. Please do NOT assume the code is complete.)

```
(very rough procedures, not precise)

binarySearch(A, key, /* you will figure out */)
  if /* (check stop condition) */
    not found
  else
    find the middle
    if A[middle] is the key
      found
    else if key is less than A[middle]
      apply binary search to the left-half
    else
      apply binary search to the right-half
```

(Notes from the live demo or live coding. Please do NOT assume the code is complete.)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\frac{n}{2}) + \Theta(1) & \text{if } n > 1 \end{cases}$$

(We used the whiteboard to derive)

$$\Theta(log_2(n))$$

Do Now Exercise

To prepare you for the lecture today, please do the following exercise.

Write the asymptotic running time of finding an item

performed on each of the data structures that we have learned so far.

Do Now Exercise

Students' answers:

Hash tables

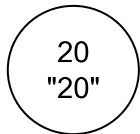
Hash tables

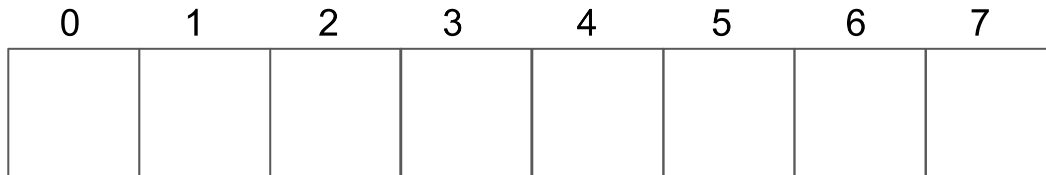
- put(key, value)
- get(key)
- remove(key)

(key-value storage)

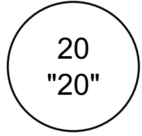
0	1	2	3	4	5	6	7

put(20, "20");

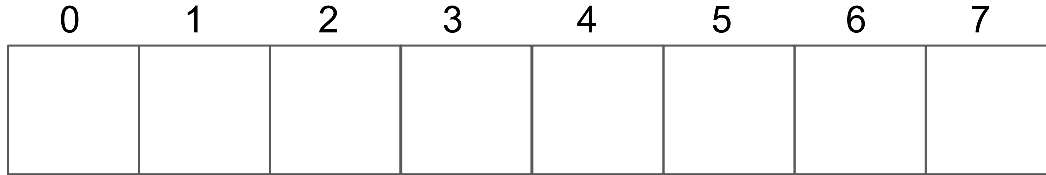




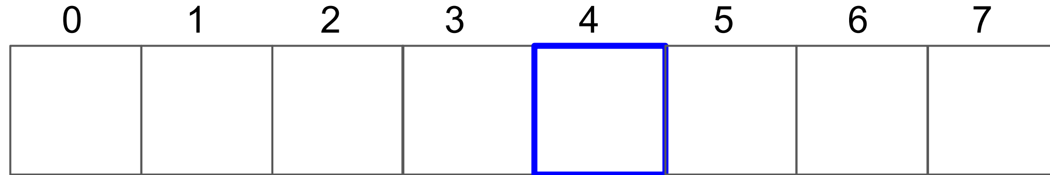
put(20, "20");



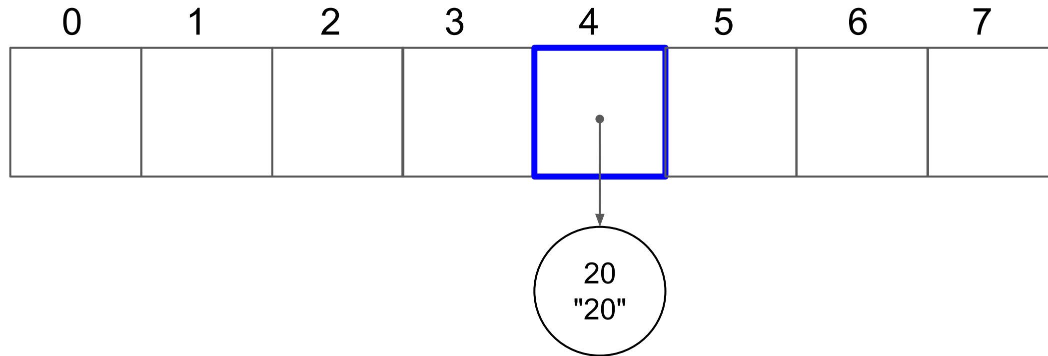
20 % 8 = 4

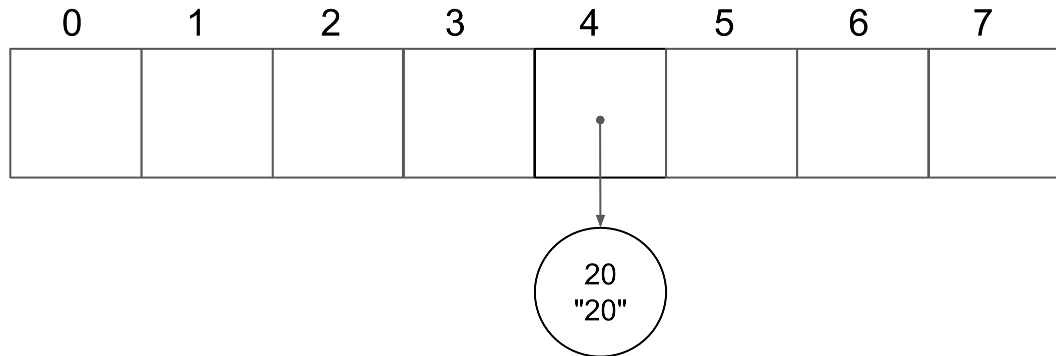


put(20, "20");



put(20, "20");




0
1
2
3
4
5
6
7
20
"20"

put(2, "2");

2
"2"

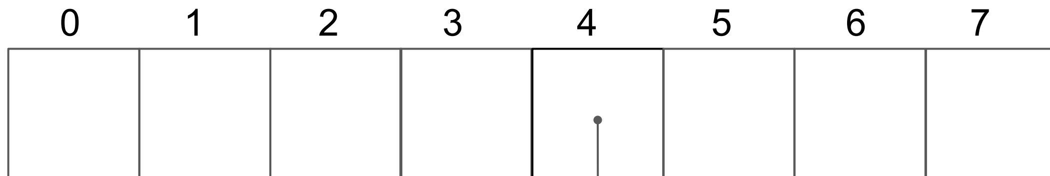
0 1 2 3 4 5 6 7

20
"20"

put(2, "2");

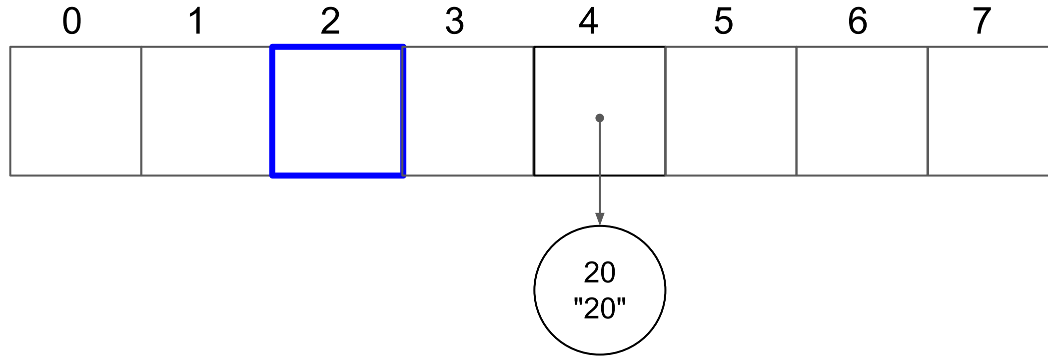
2
"2"

2 % 8 = 2

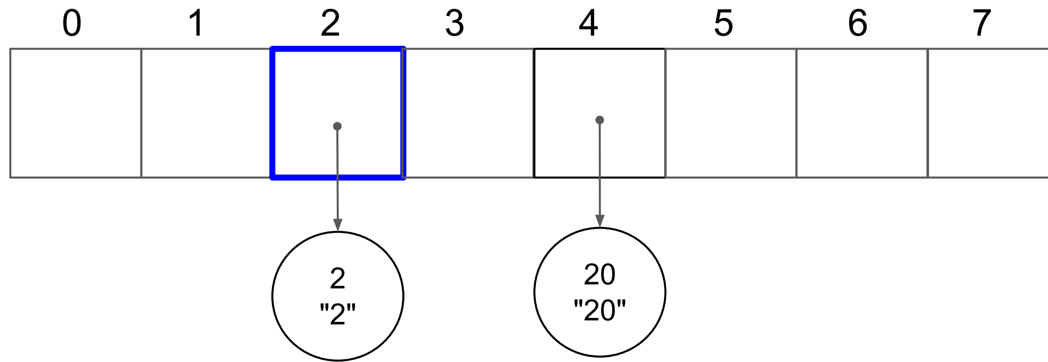


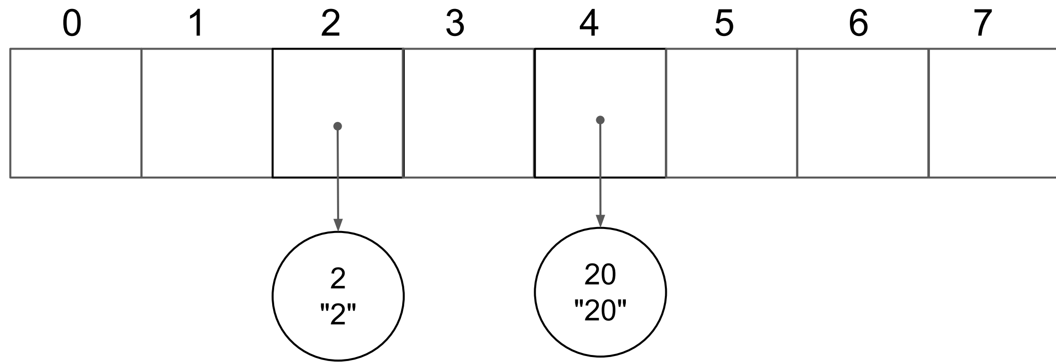
20
"20"

put(2, "2");

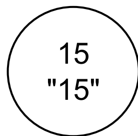


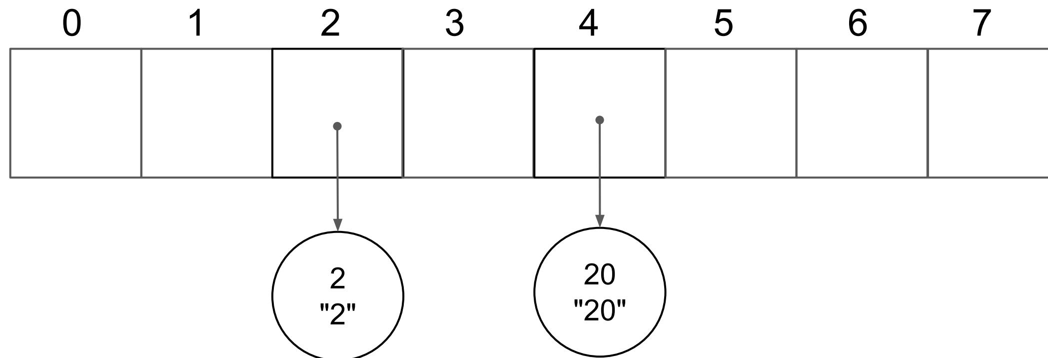
put(2, "2");



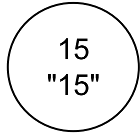

0
1
2
3
4
5
6
7
2
"2"
20
"20"

put(15, "15");

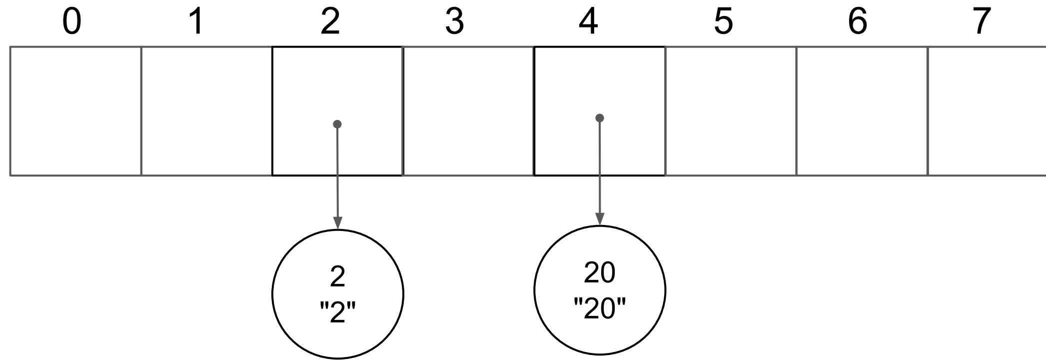




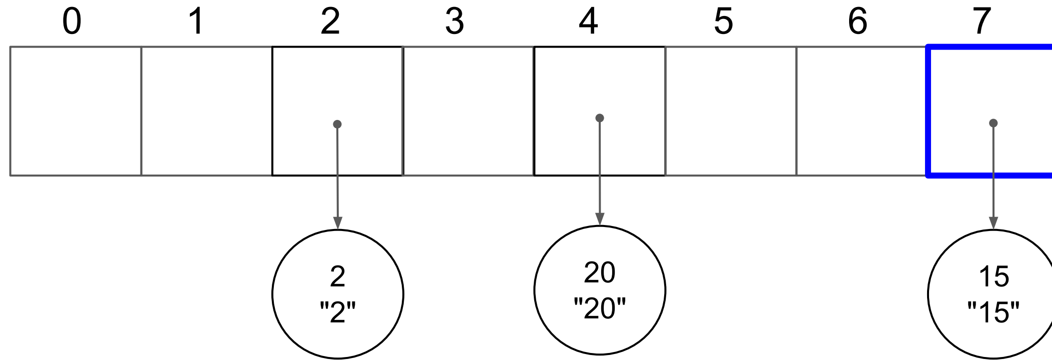
put(15, "15");

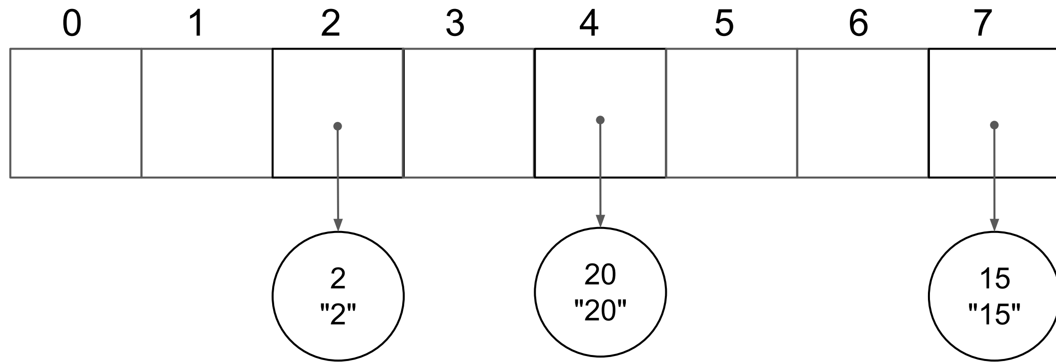


15 % 8 = 7

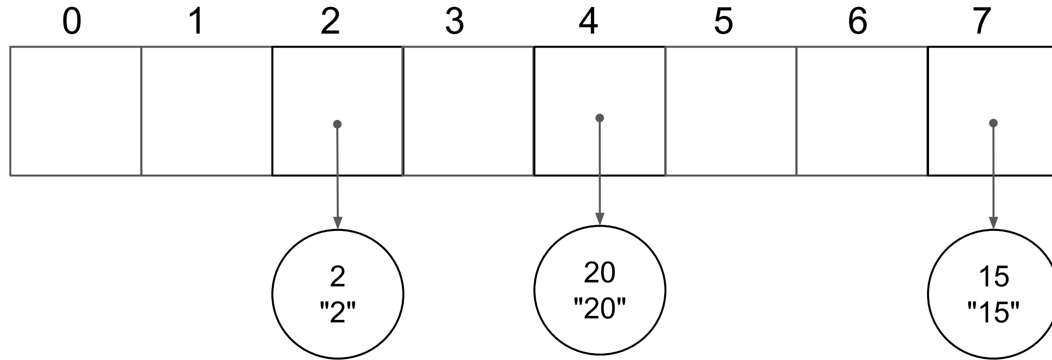


put(15, "15");



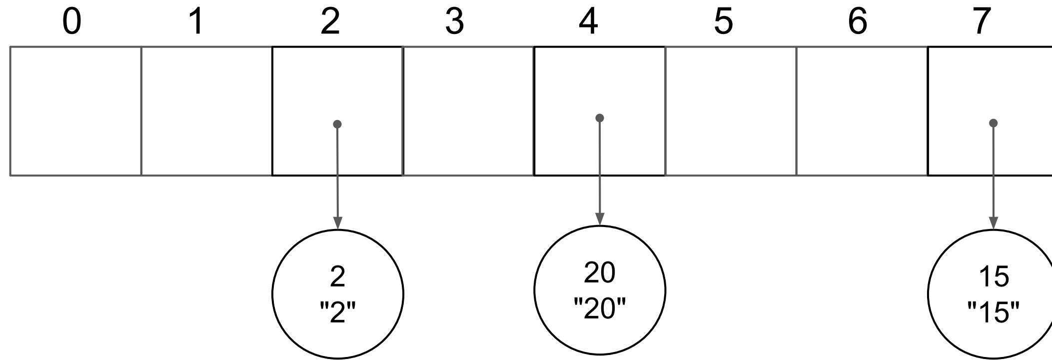

0
1
2
3
4
5
6
7
2
"2"
20
"20"
15
"15"

get(20);

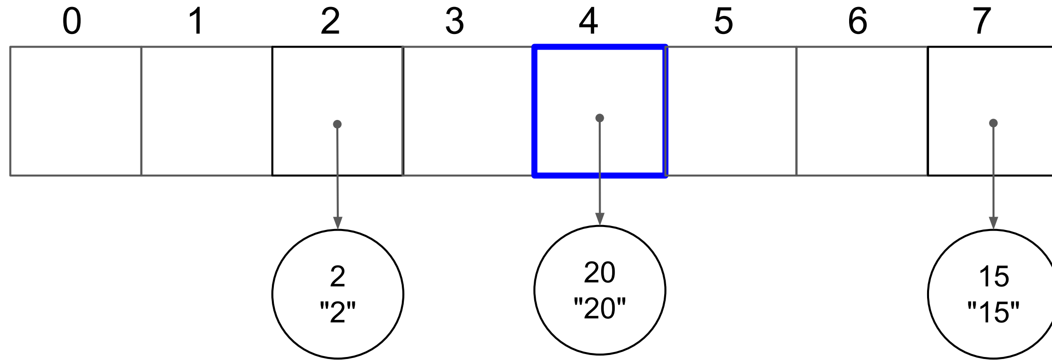


get(20);

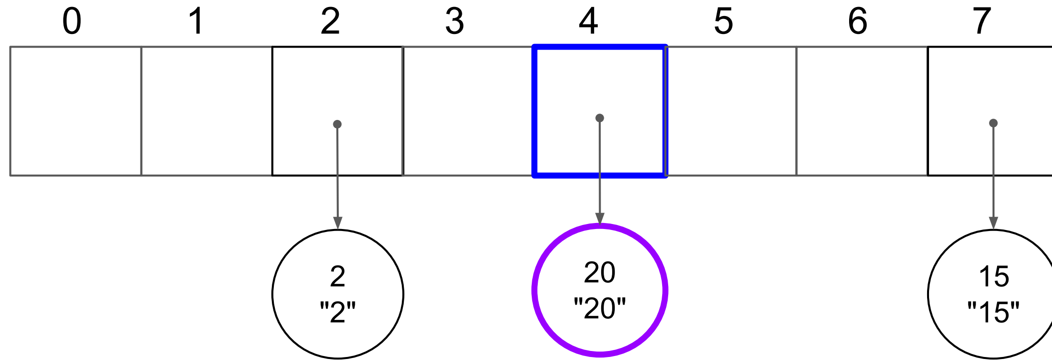
20 % 8 = 4



get(20);



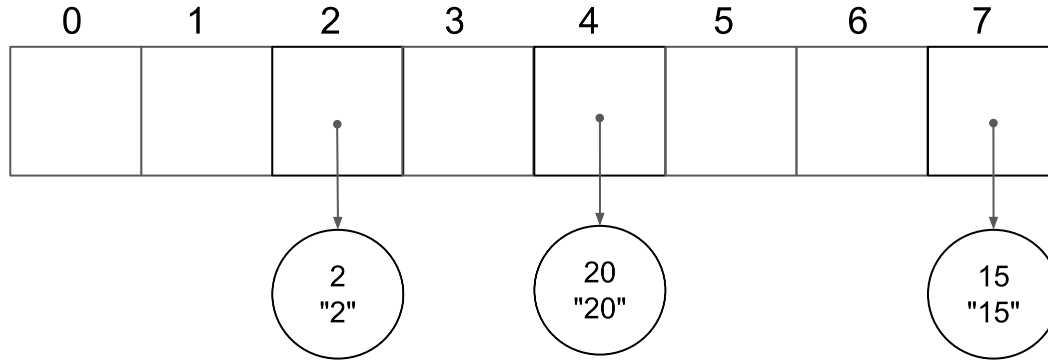
get(20);



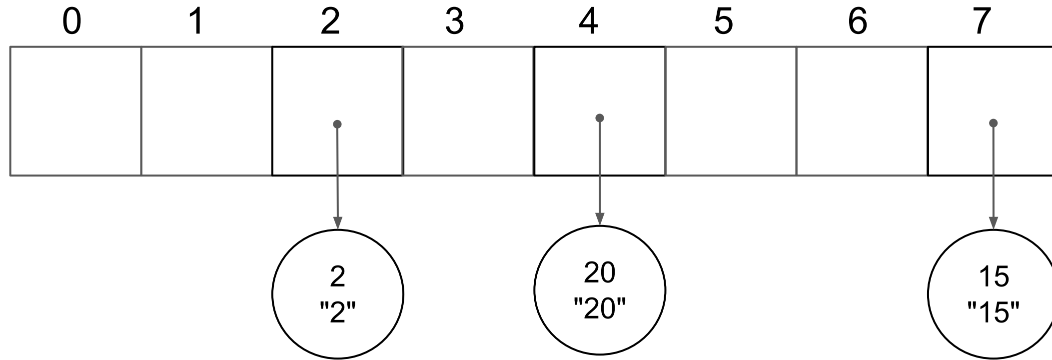
get(20);

"20"

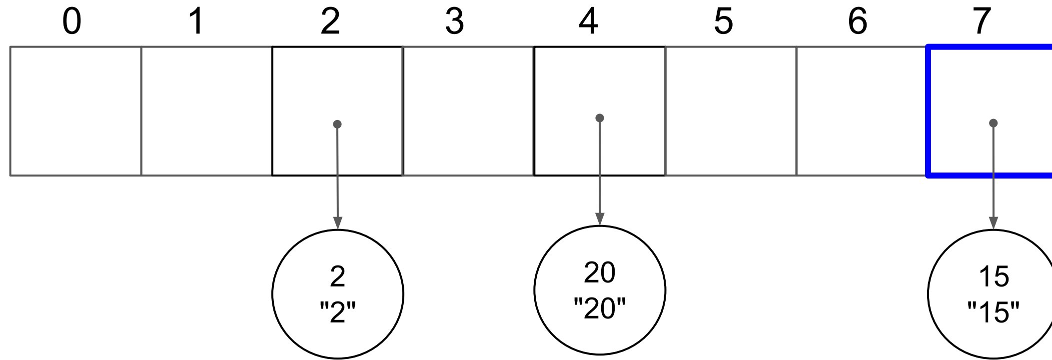
20 % 8 = 4



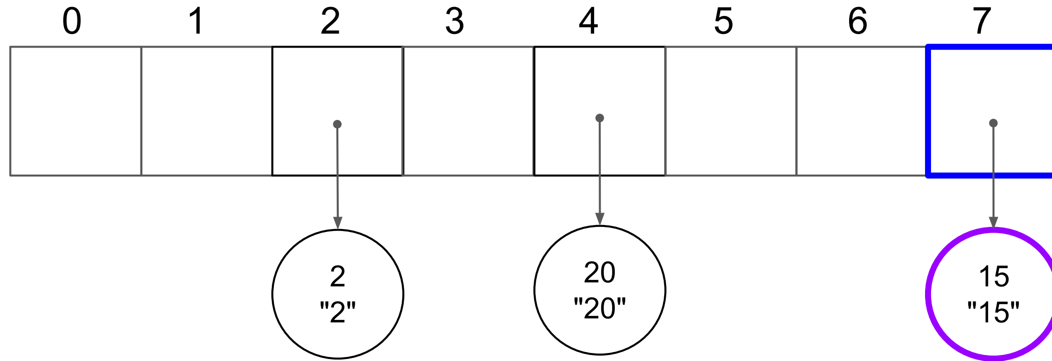
get(15);



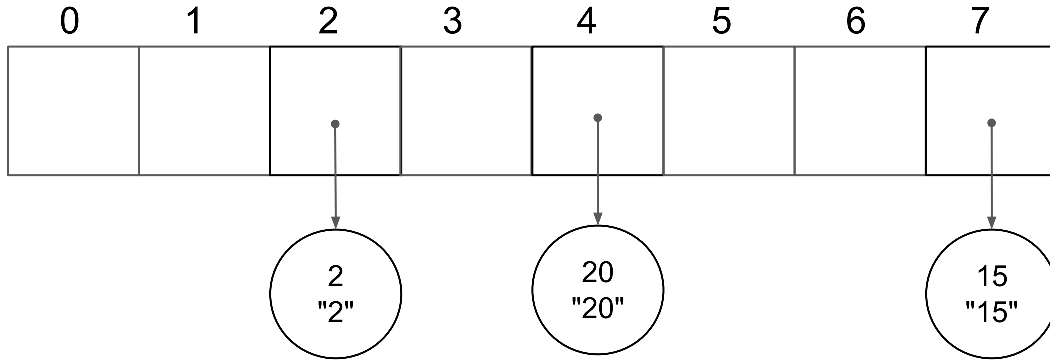
get(15);

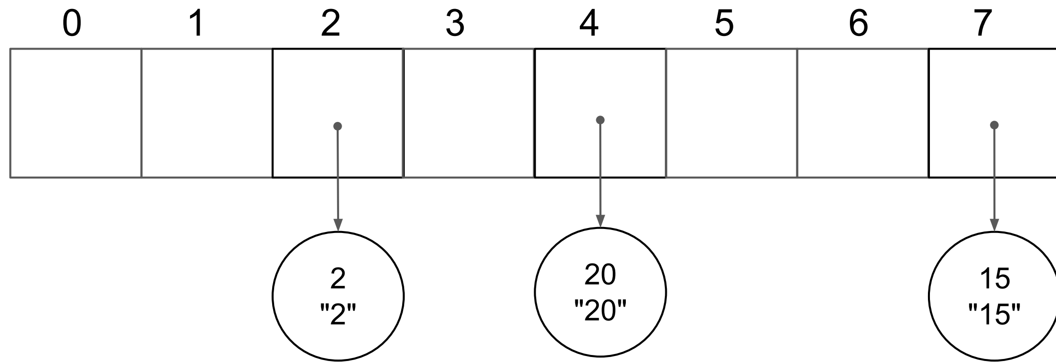


get(15);

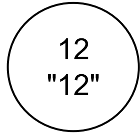


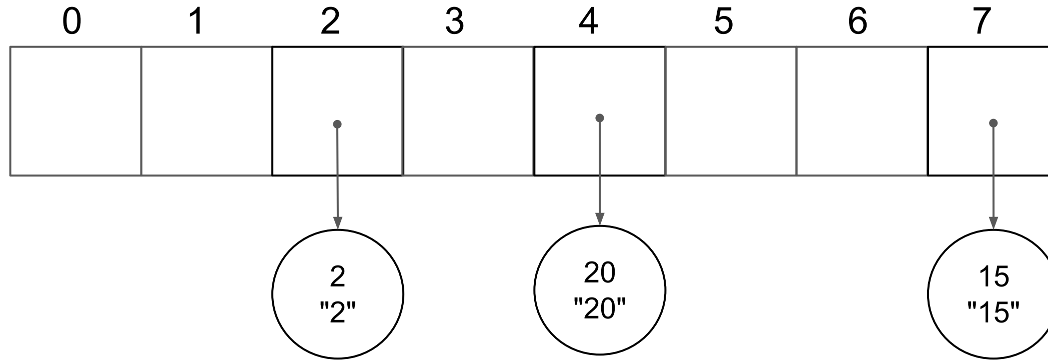
get(15);



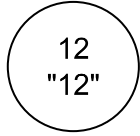

0
1
2
3
4
5
6
7
2
"2"
20
"20"
15
"15"

put(12, "12");

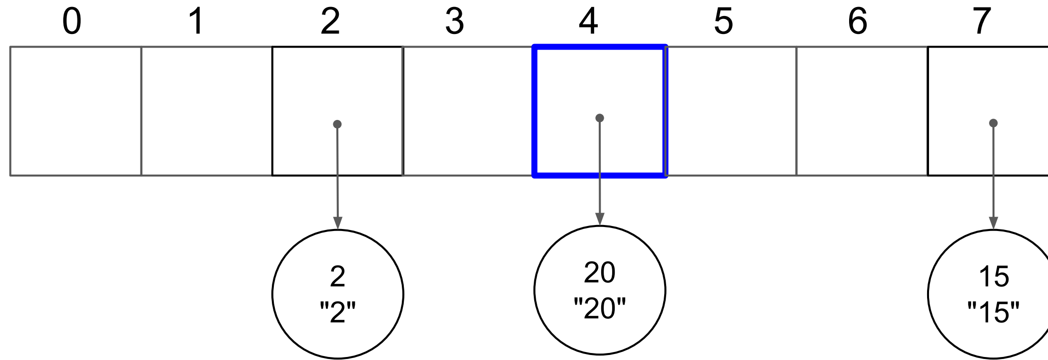




put(12, "12");

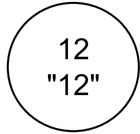


12 % 8 = 4

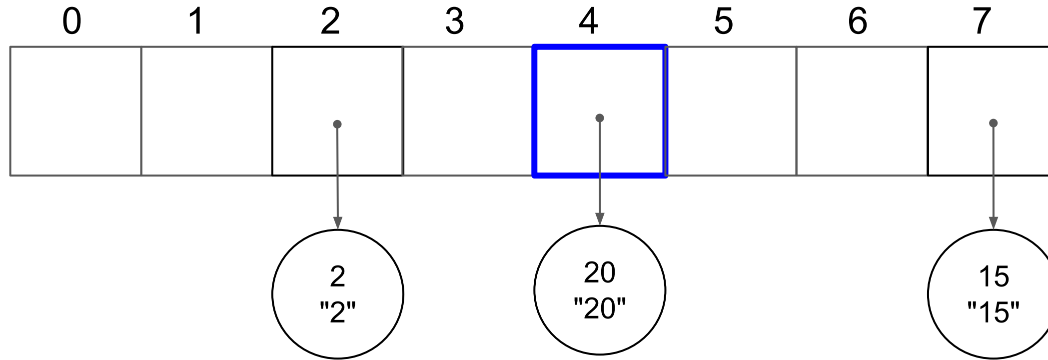


Collision

put(12, "12");

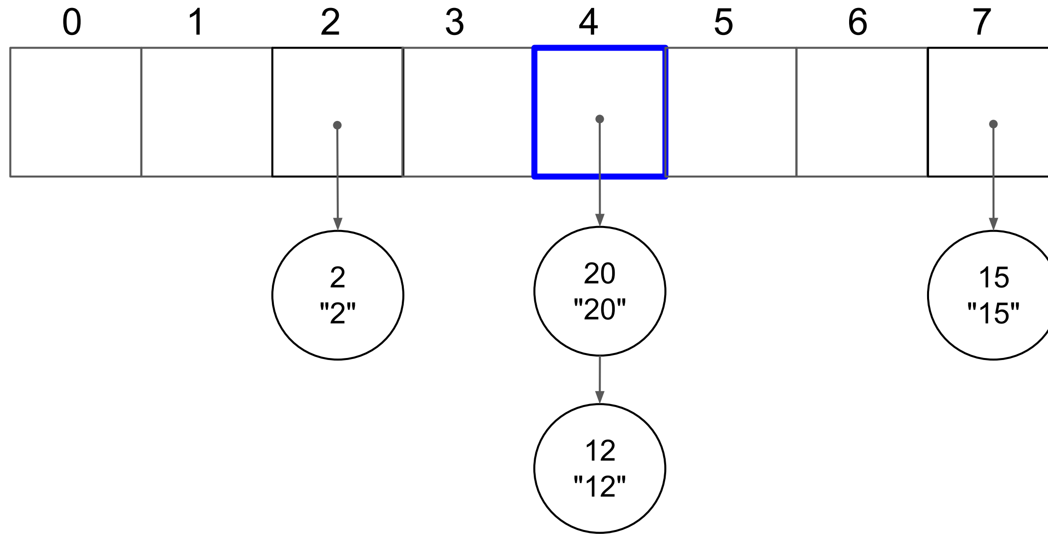


12 % 8 = 4



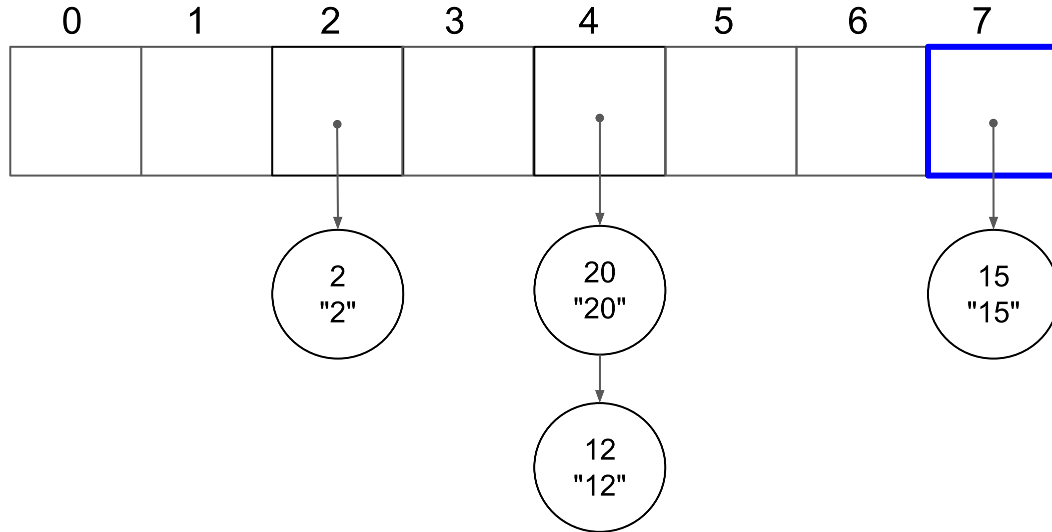
put(12, "12");

12 % 8 = 4



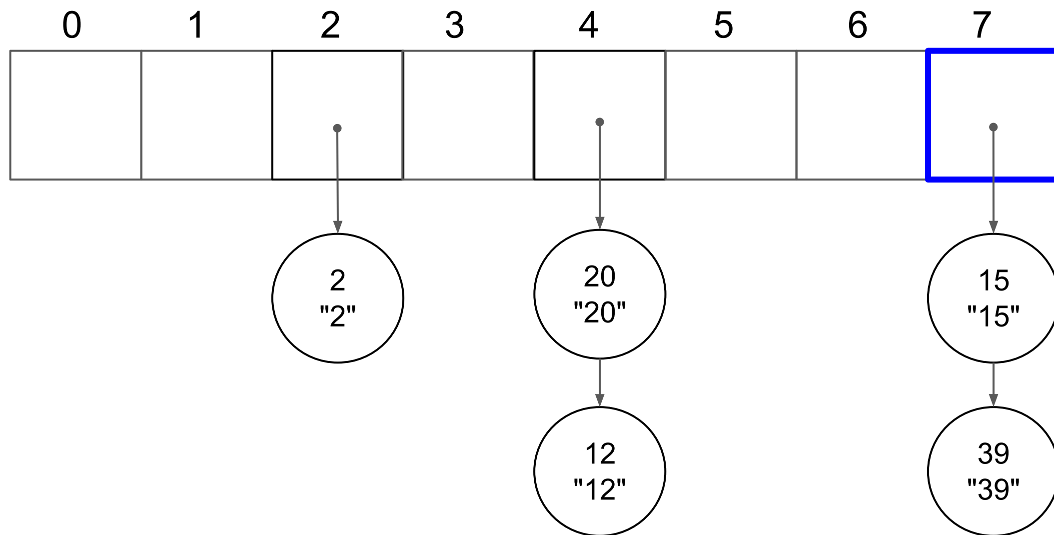
Chaining

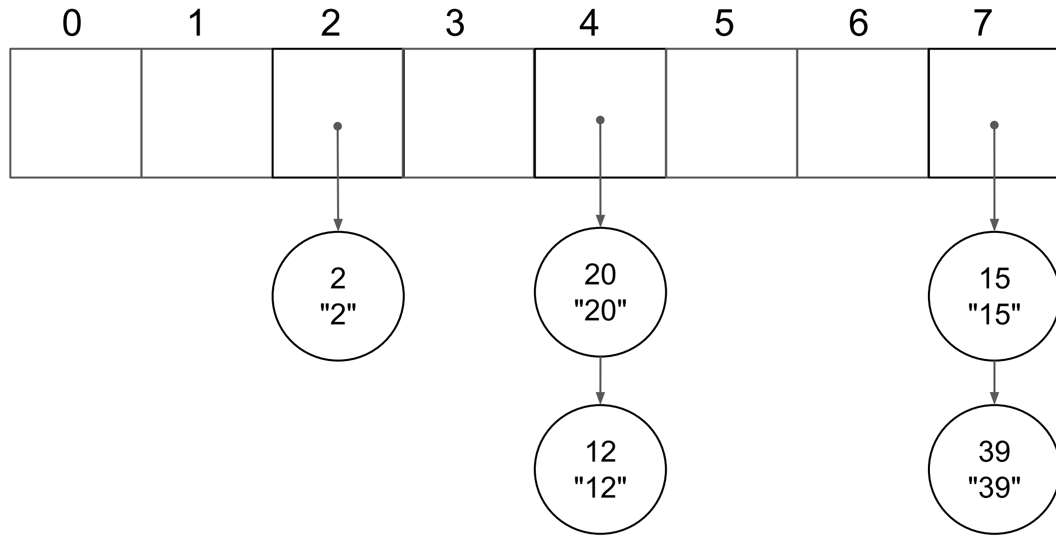
put(39, "39");



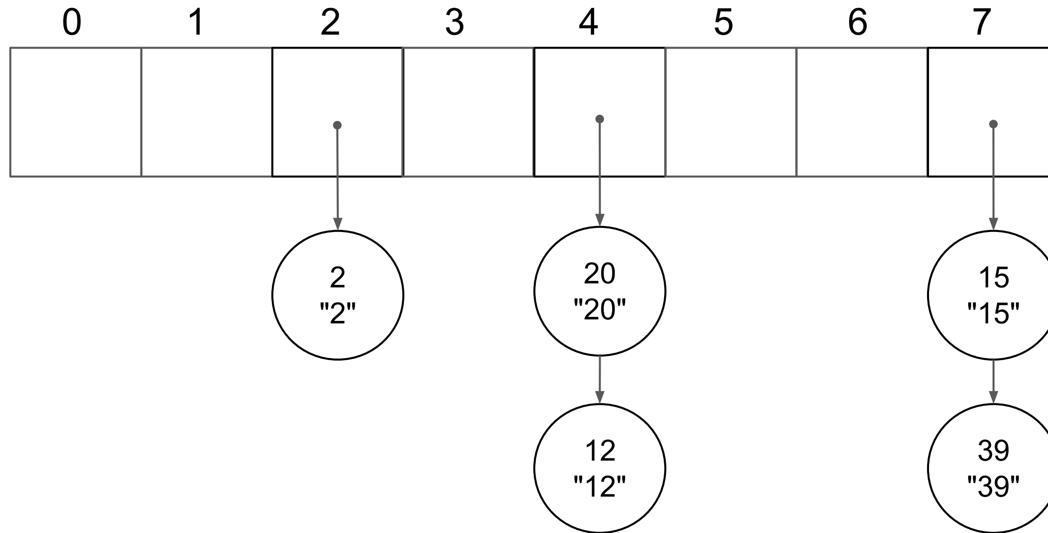
put(39, "39");

39 % 8 = 7

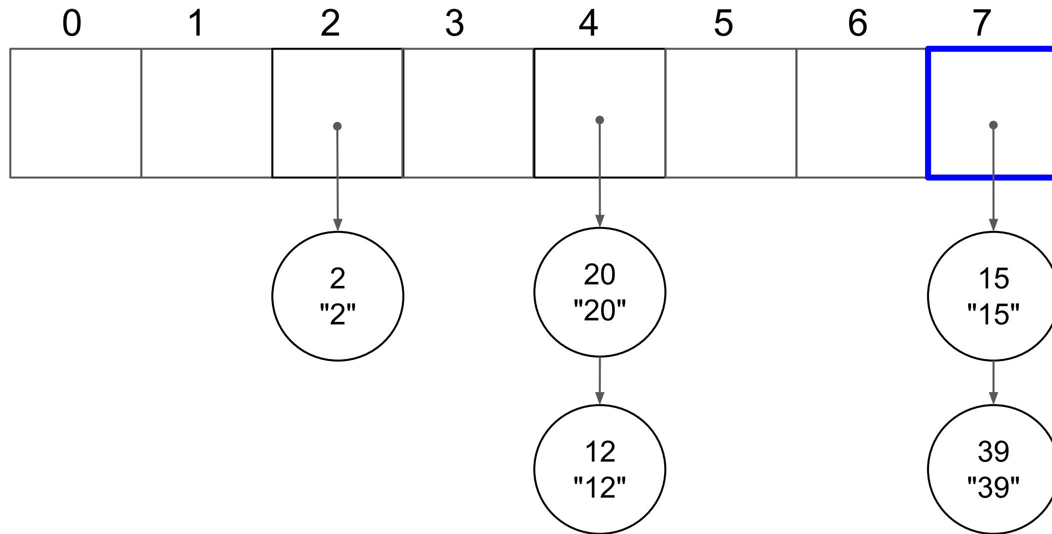



0
1
2
3
4
5
6
7
2
"2"
20
"20"
12
"12"
15
"15"
39
"39"

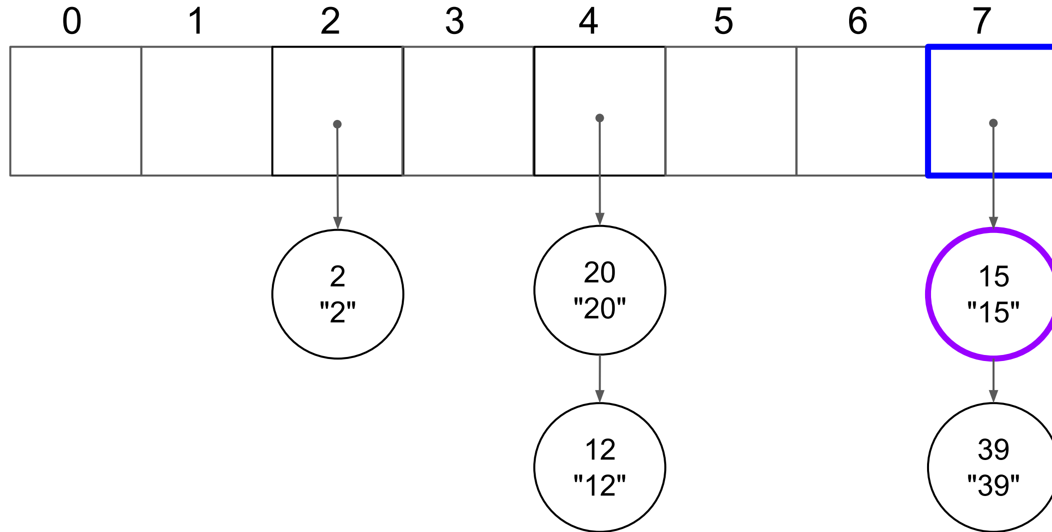
get(39);



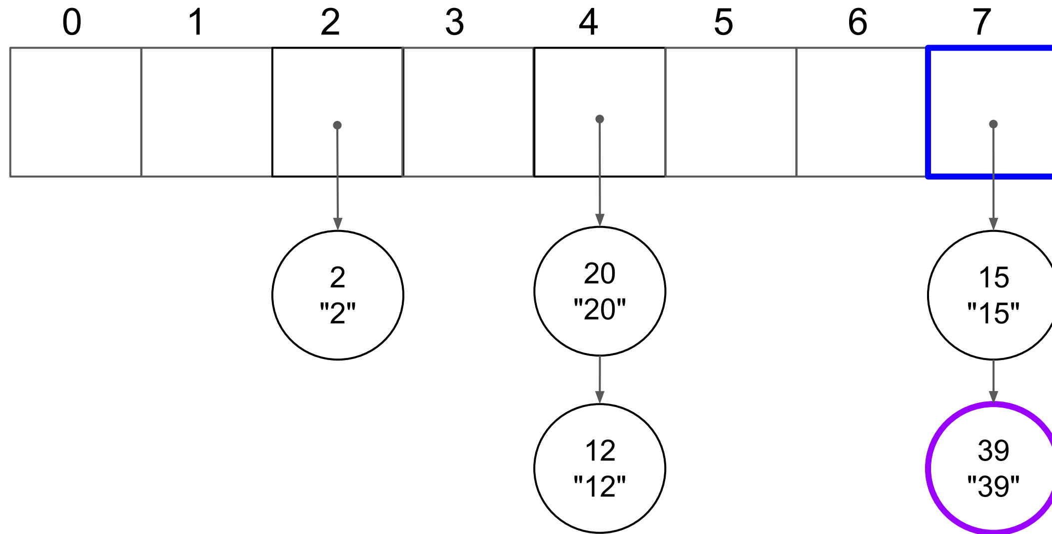
get(39);



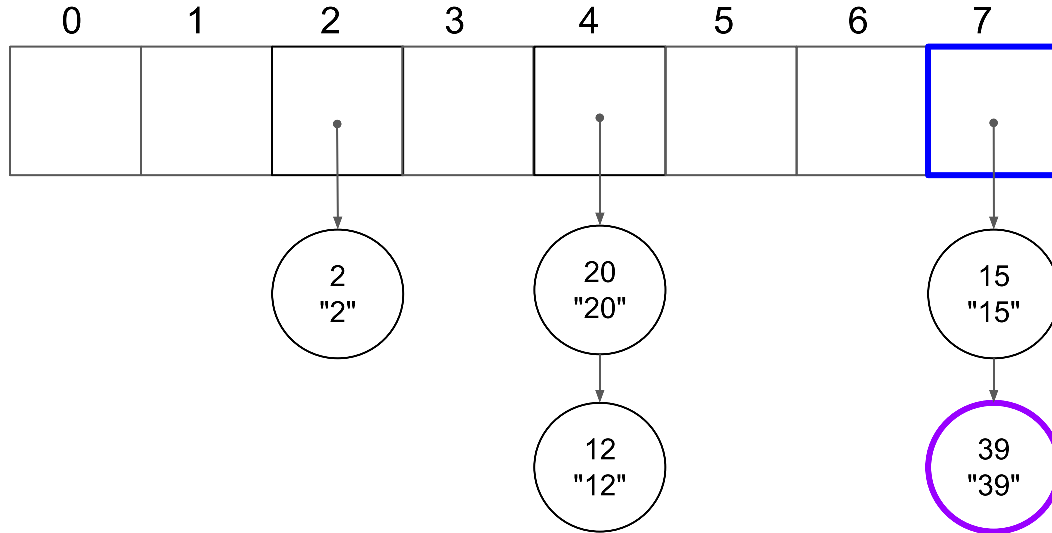
get(39);



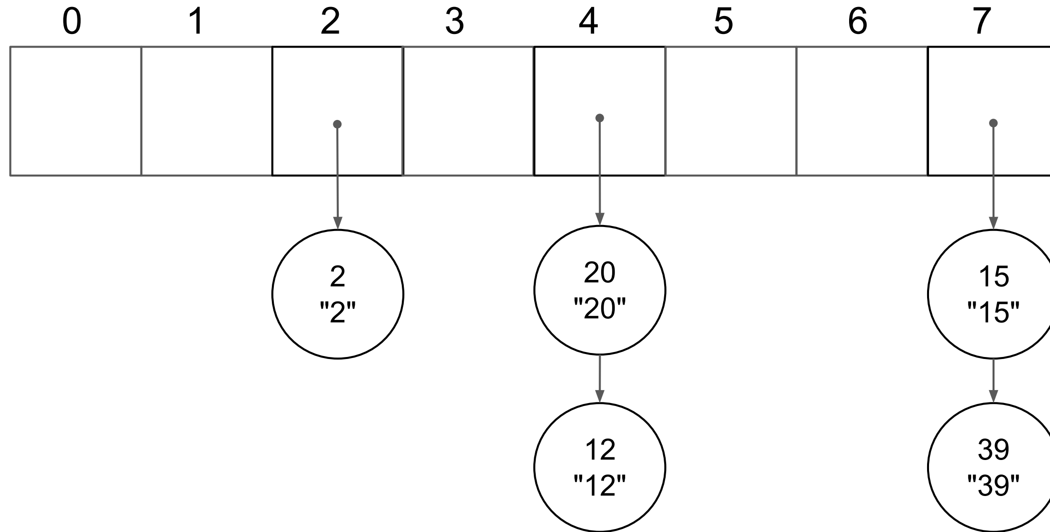
get(39);



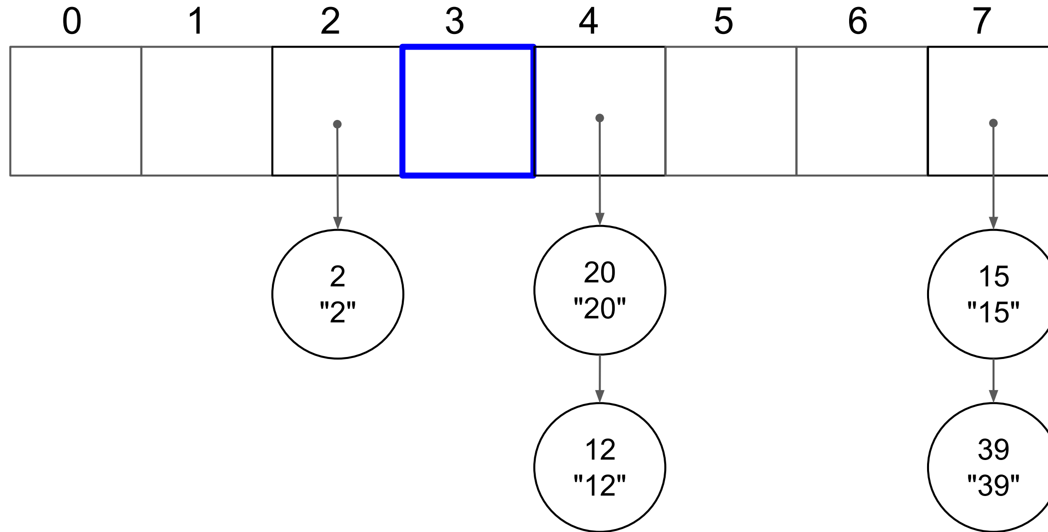
get(39);



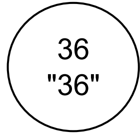
get(11);

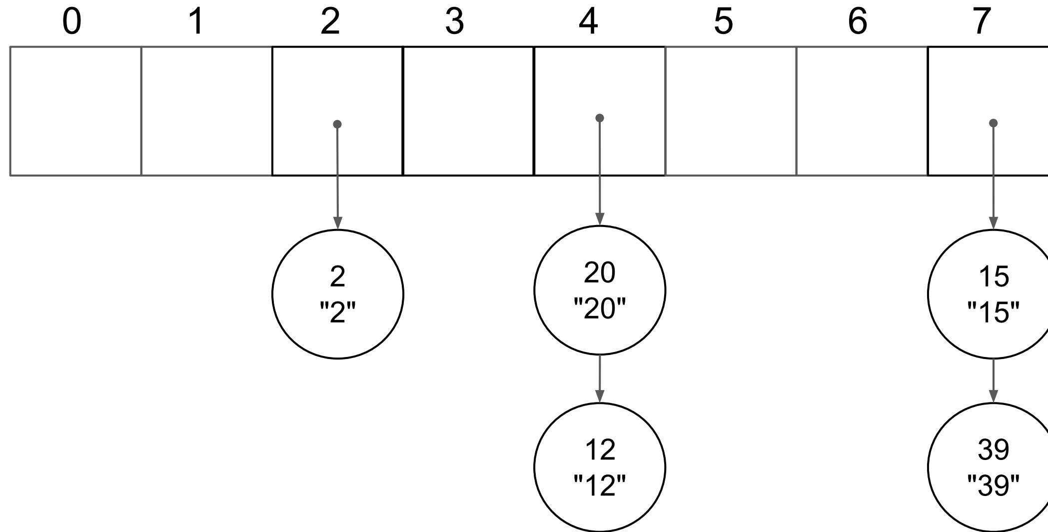


get(11);

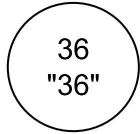


put(36, "36");

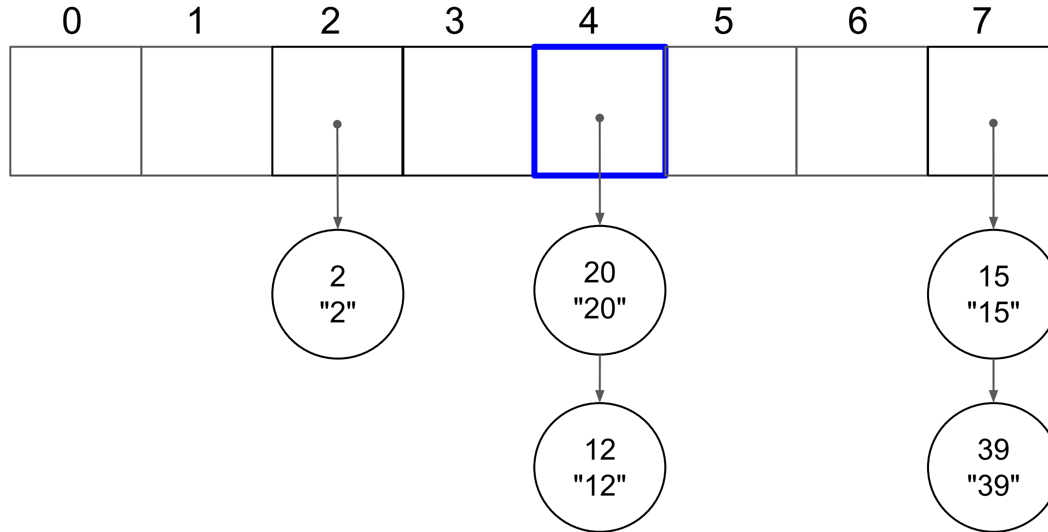




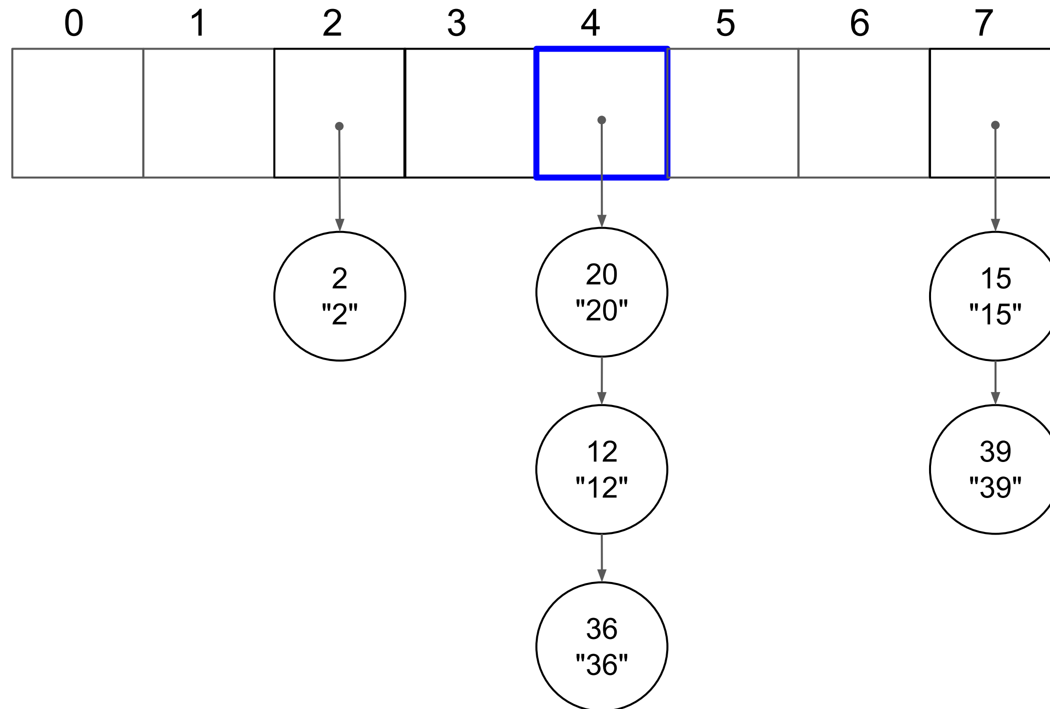
put(36, "36");

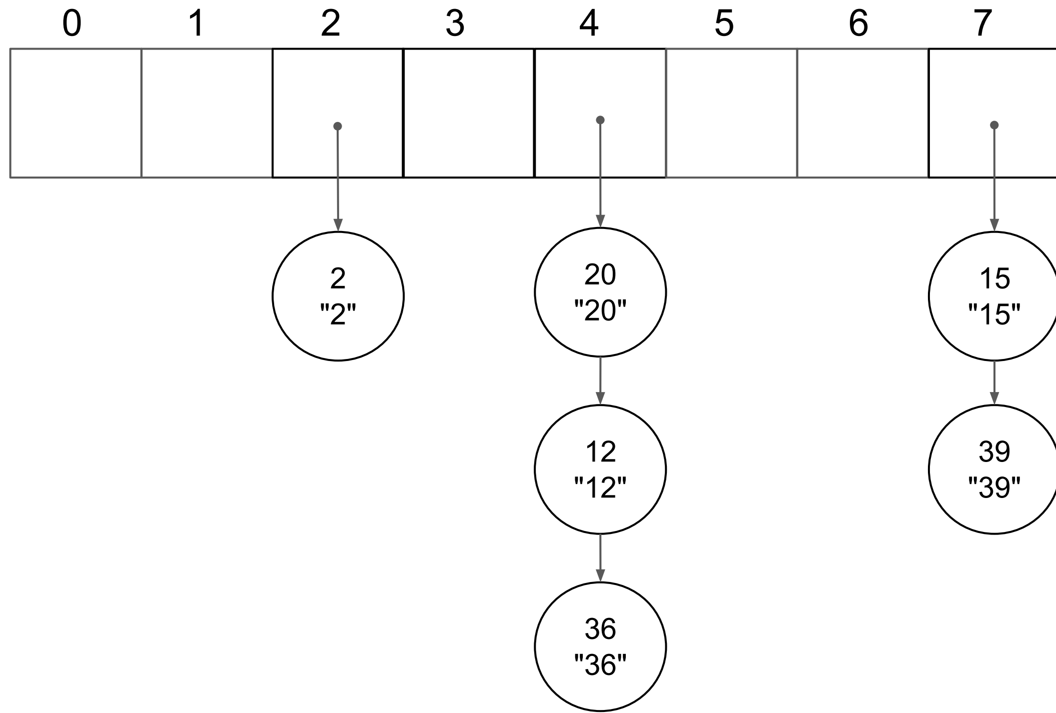


36 % 8 = 4

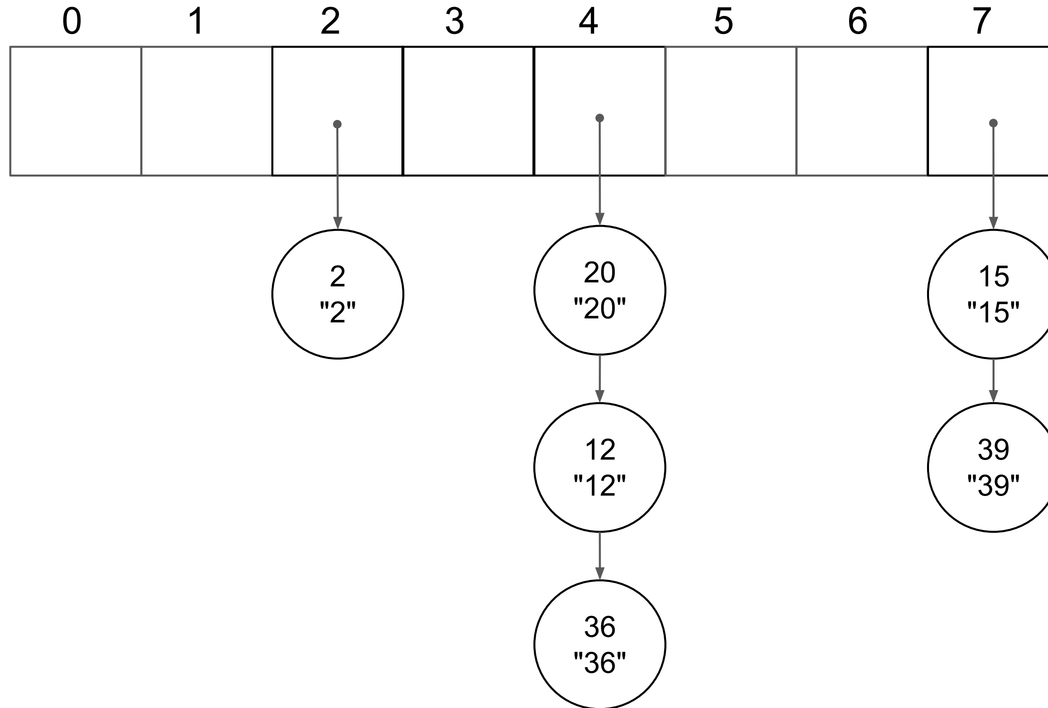


put(36, "36");

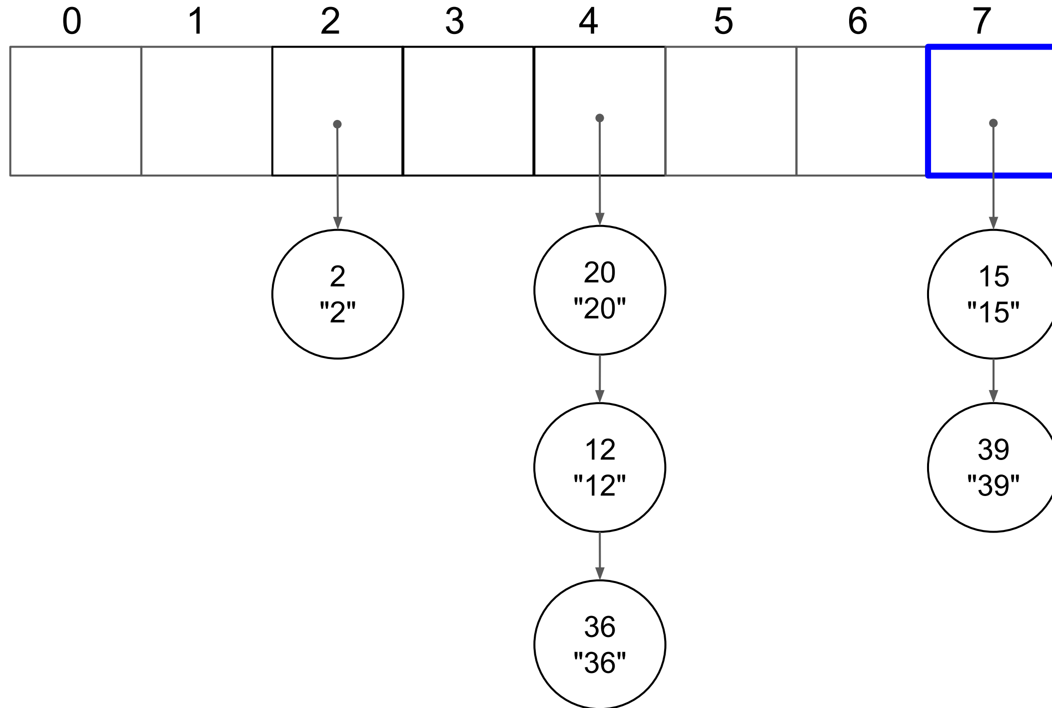



0
1
2
3
4
5
6
7
2
"2"
20
"20"
12
"12"
36
"36"
15
"15"
39
"39"

get(23);

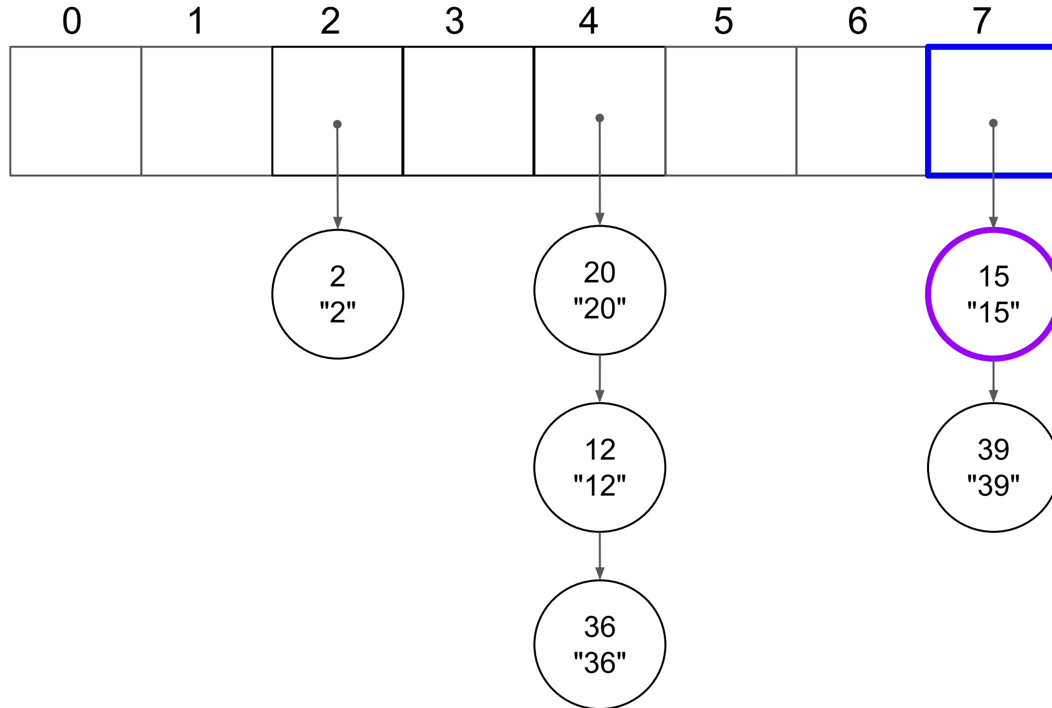


get(23);



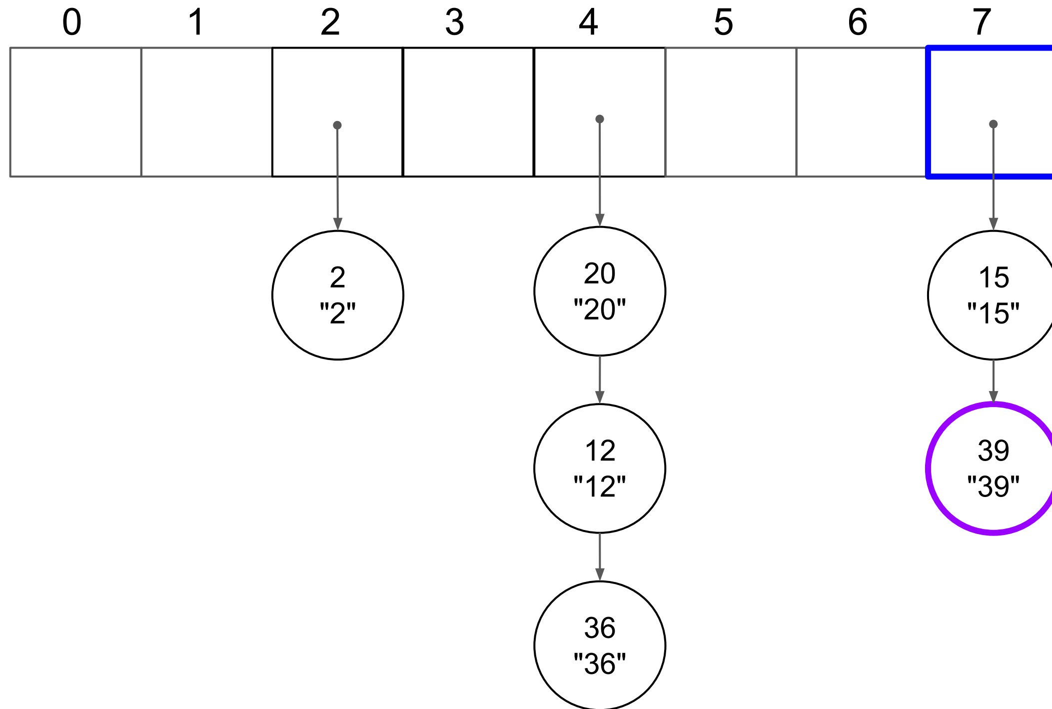
get(23);

23 % 8 = 7

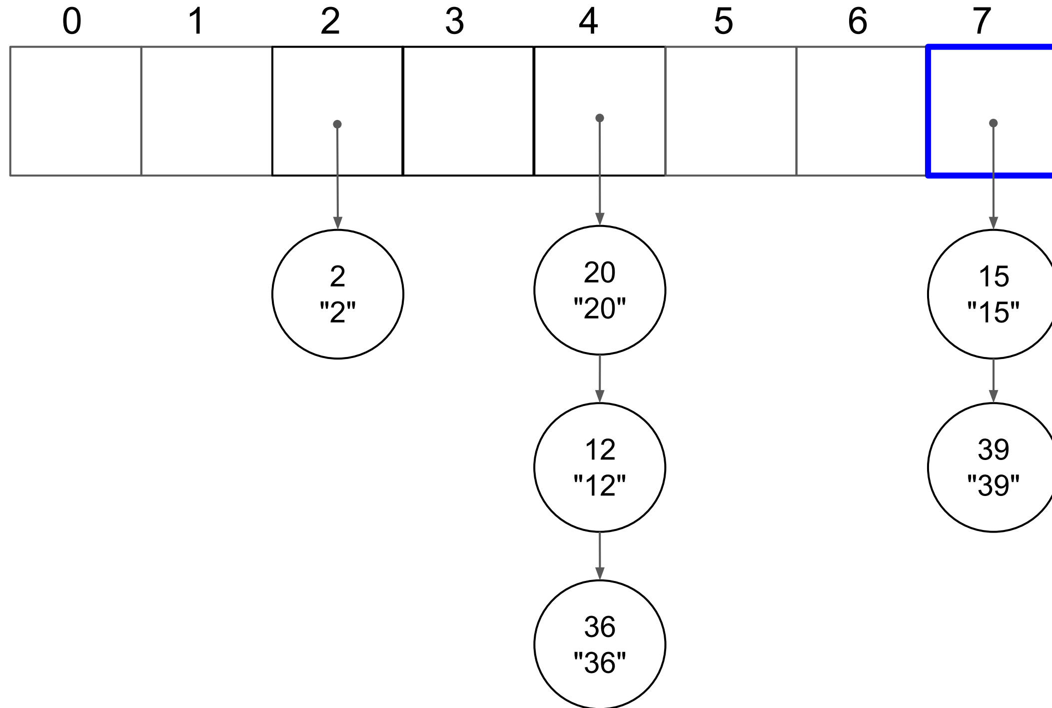


get(23);

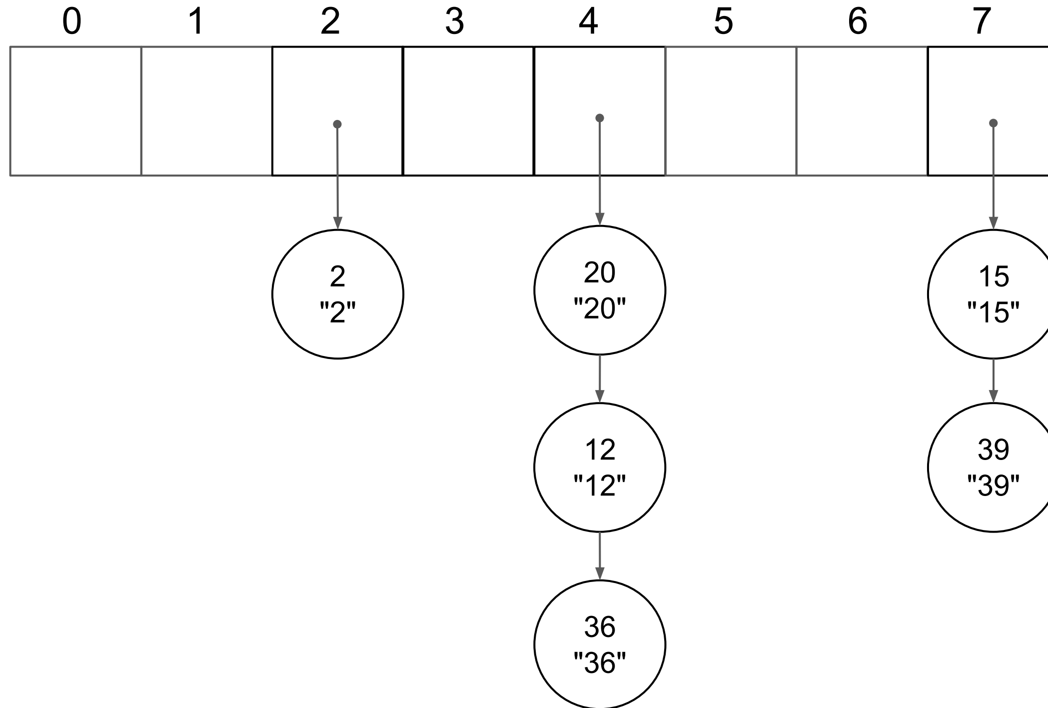
23 % 8 = 7



get(23);

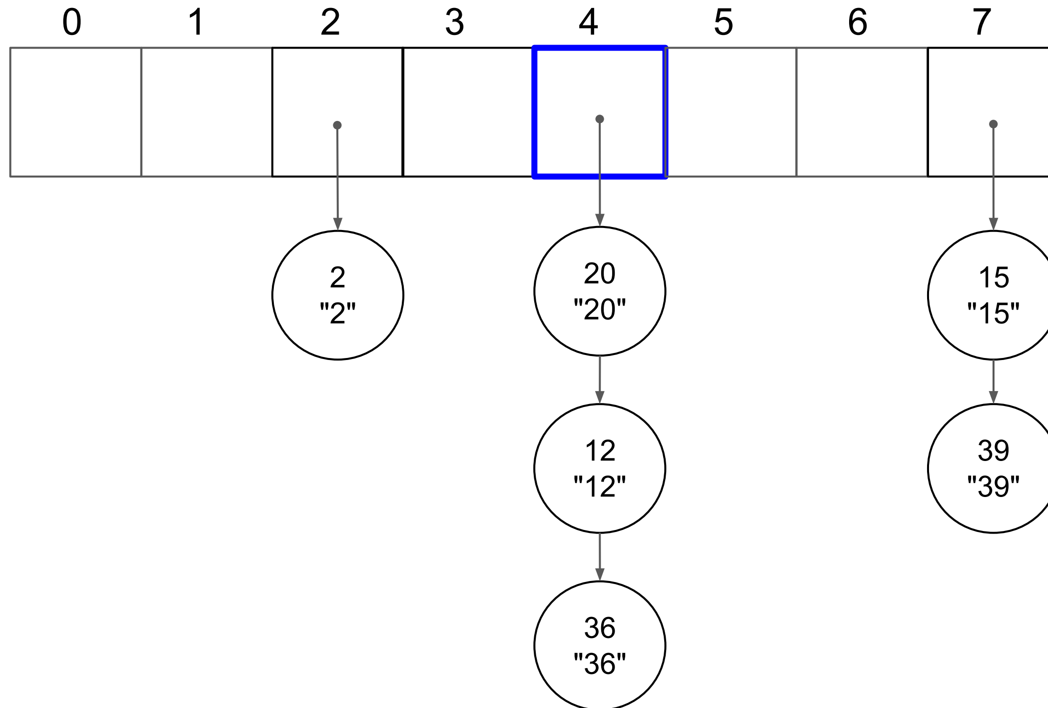


remove(36);



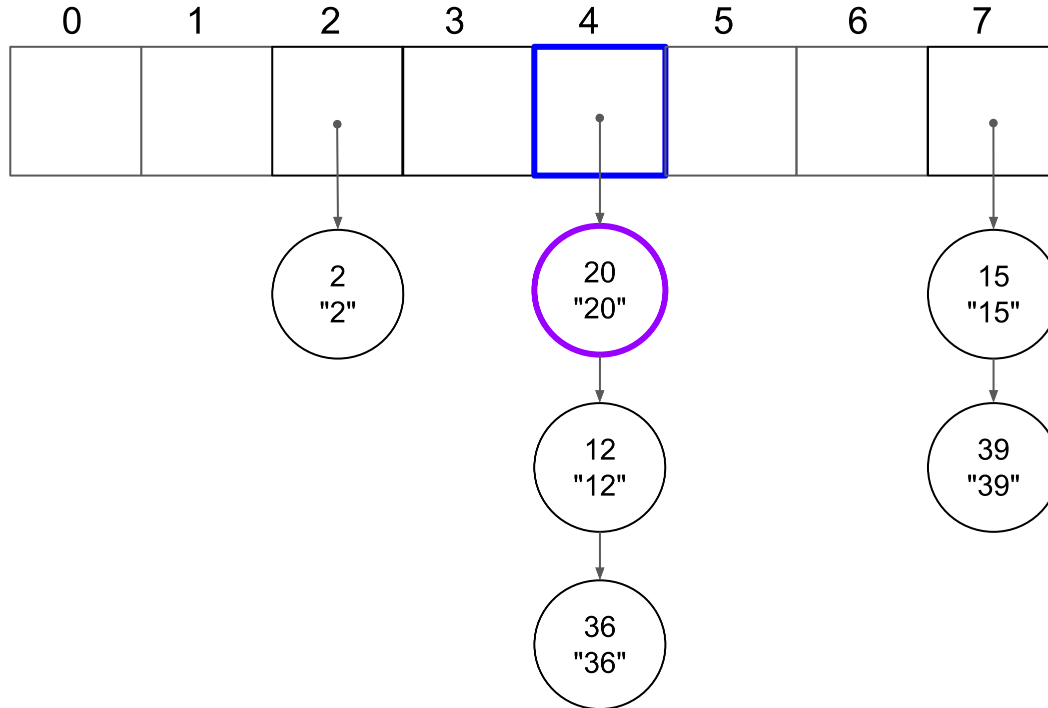
remove(36);

36 % 8 = 4

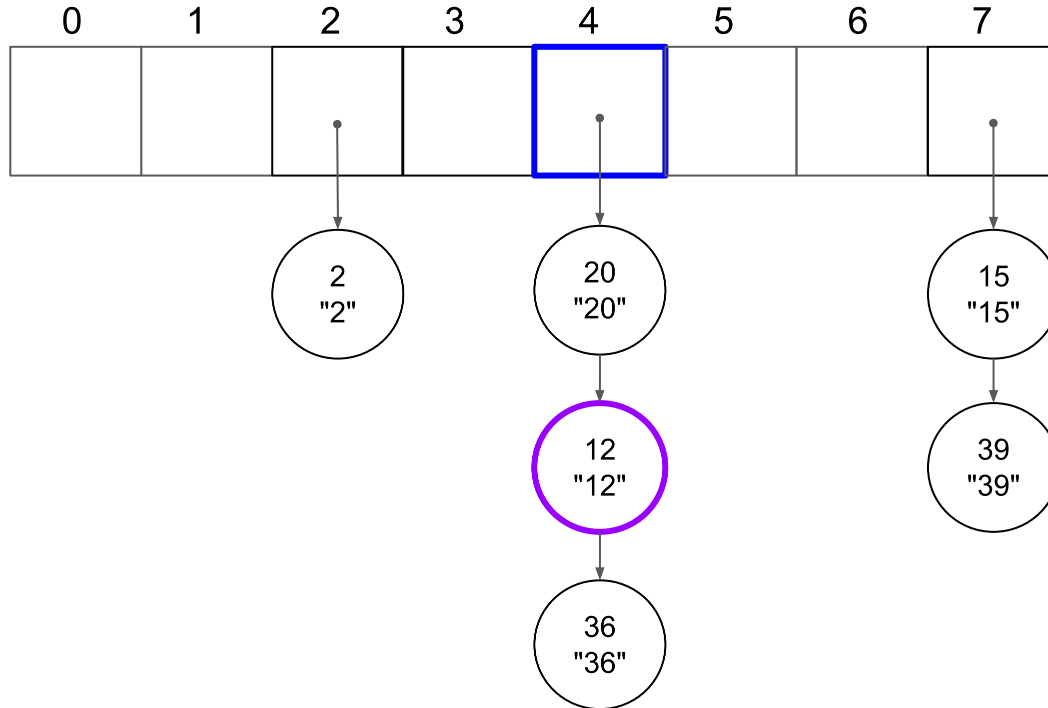


remove(36);

36 % 8 = 4

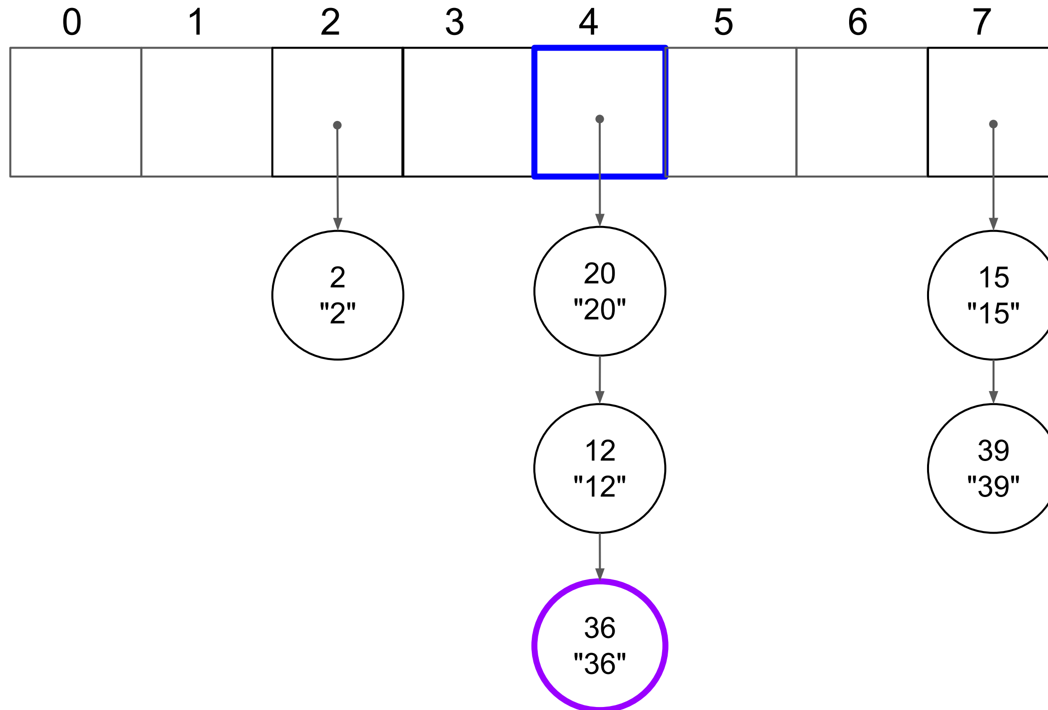


remove(36);

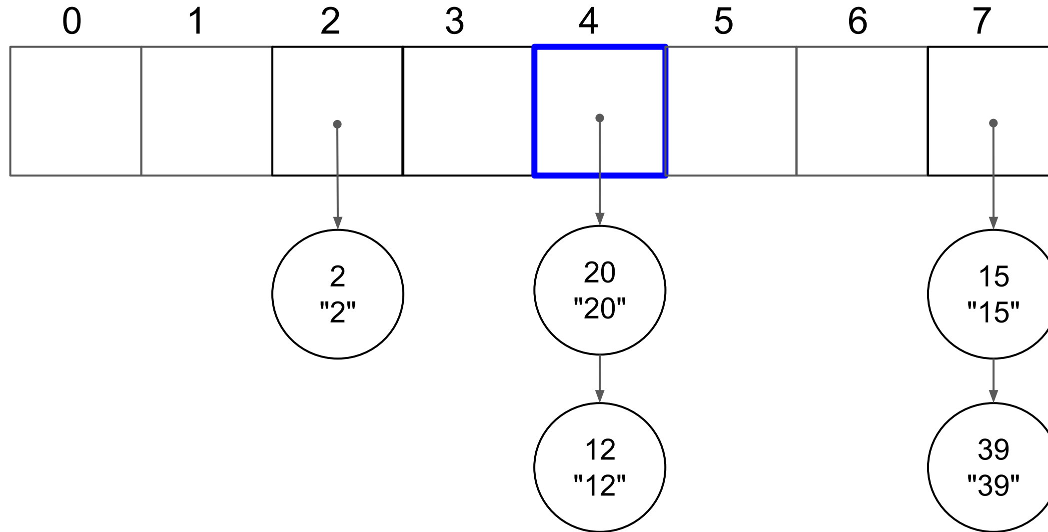


remove(36);

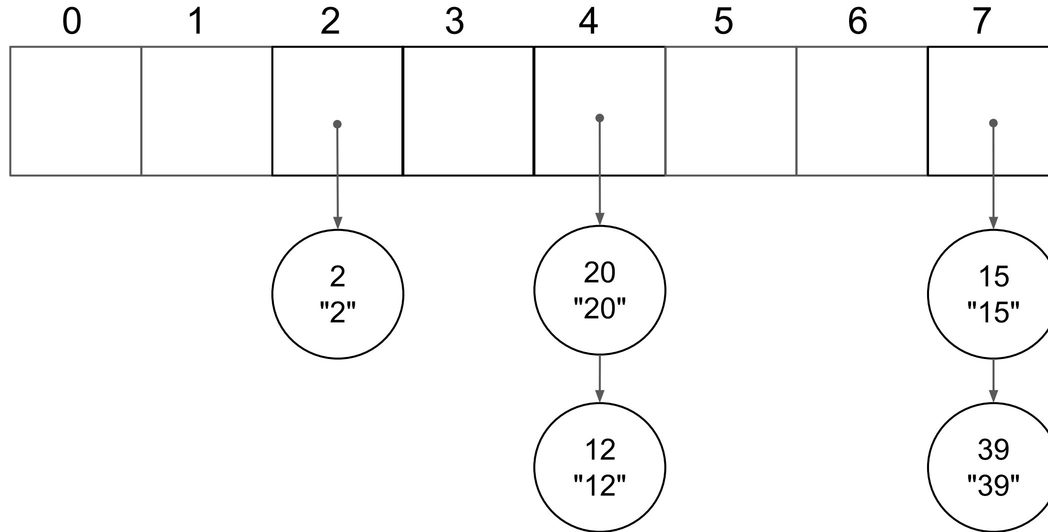
36 % 8 = 4



remove(36);

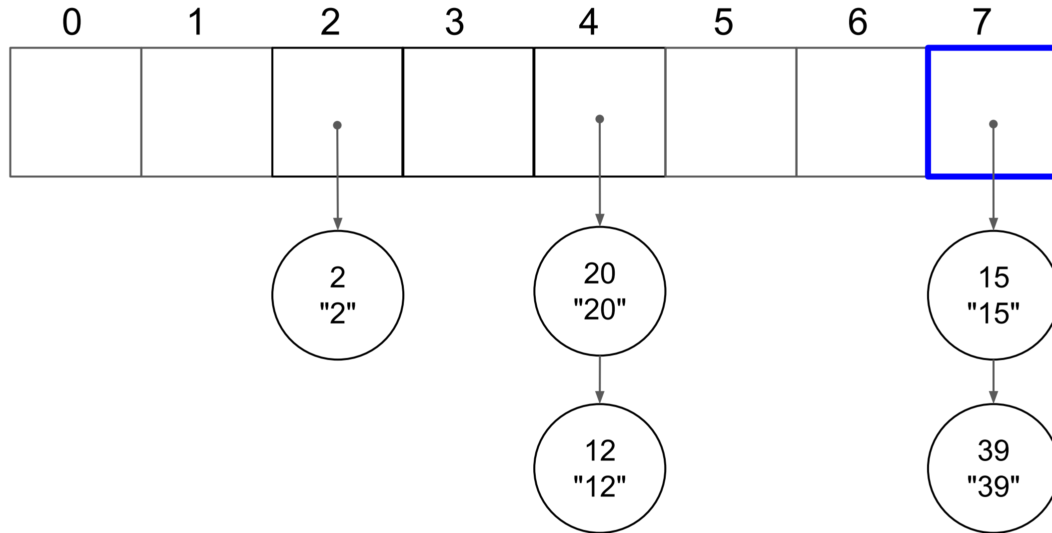


remove(15);



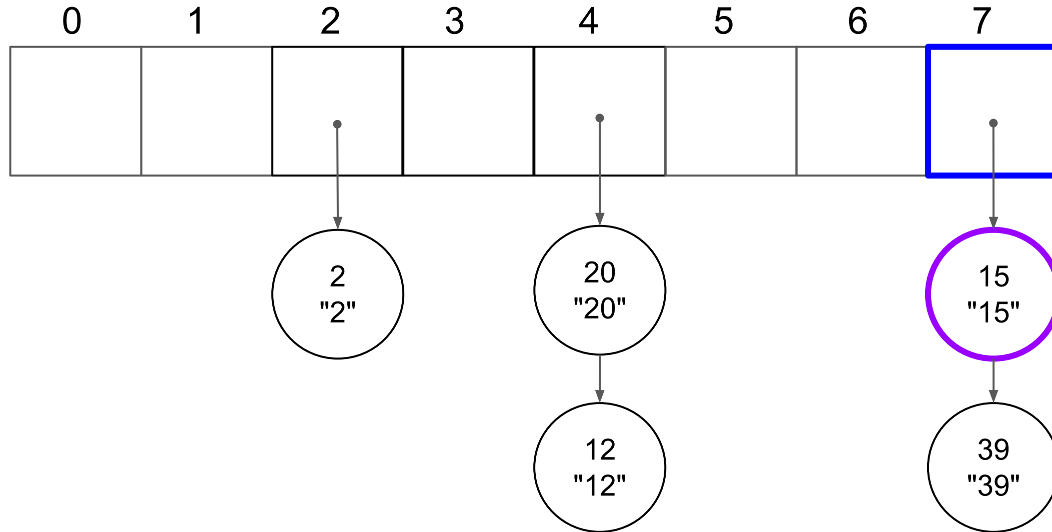
remove(15);

15 % 8 = 7

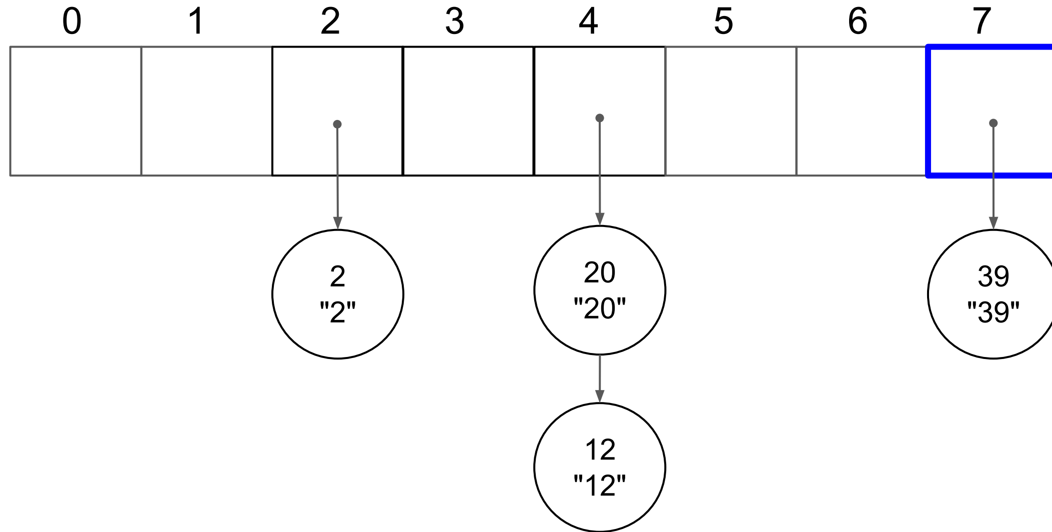


remove(15);

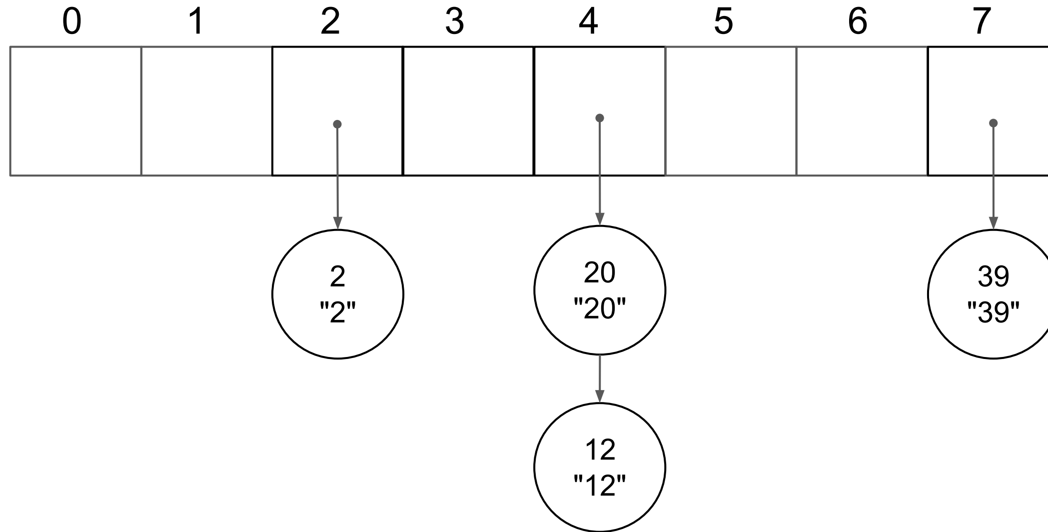
15 % 8 = 7



remove(15);

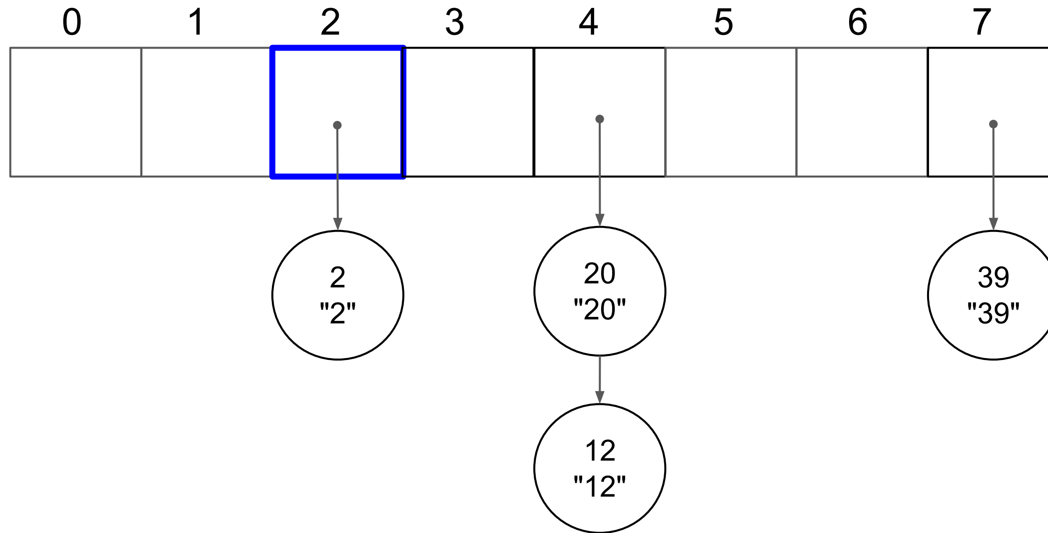


remove(18);



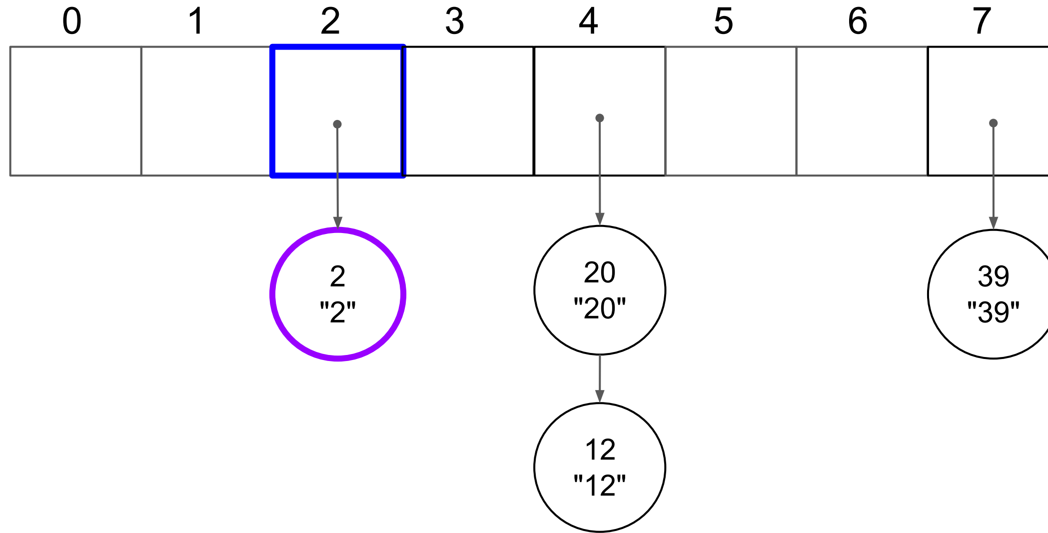
remove(18);

18 % 8 = 2

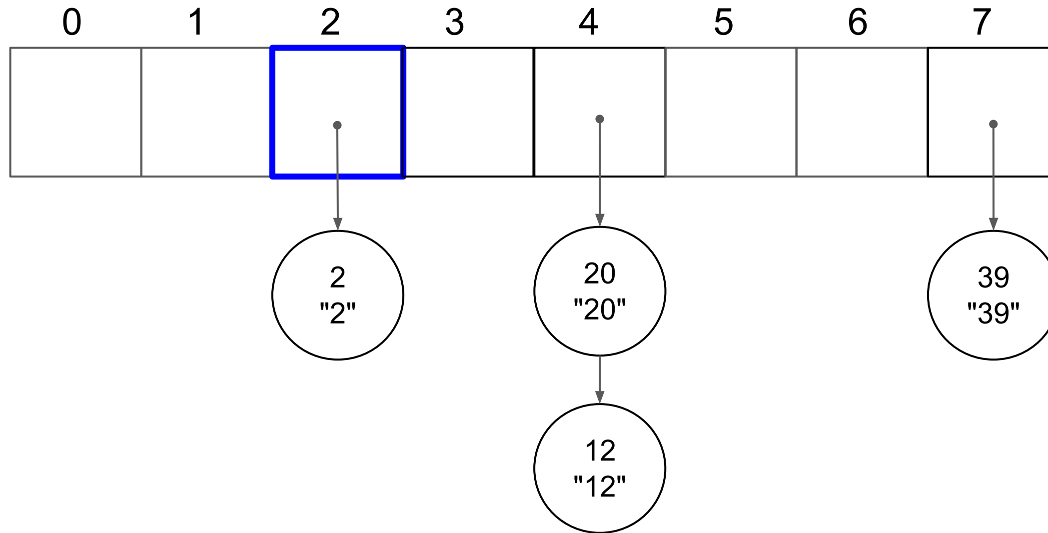


remove(18);

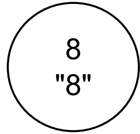
18 % 8 = 2



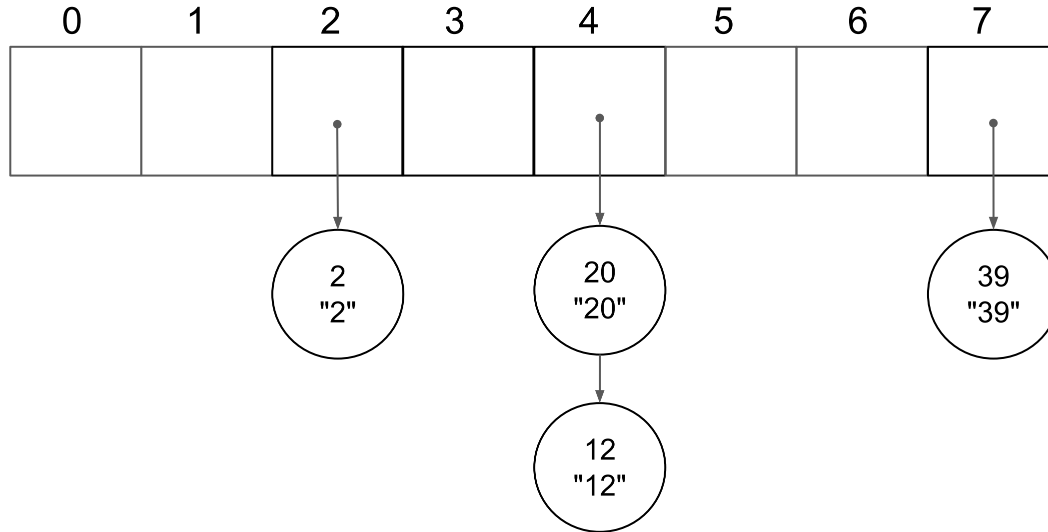
remove(18);



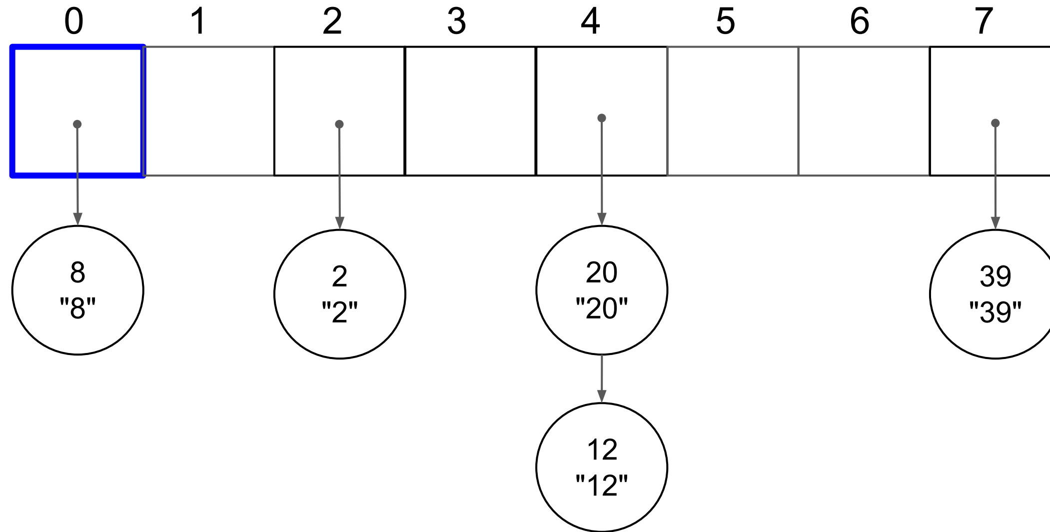
put(8, "8");



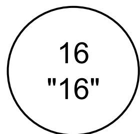
8 % 8 = 0



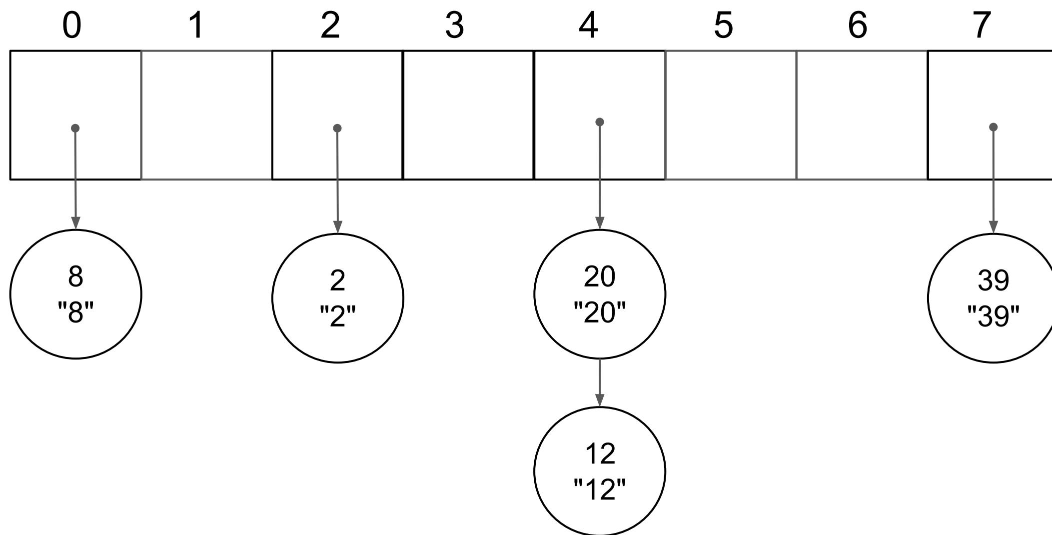
put(8, "8");



put(16, "16");

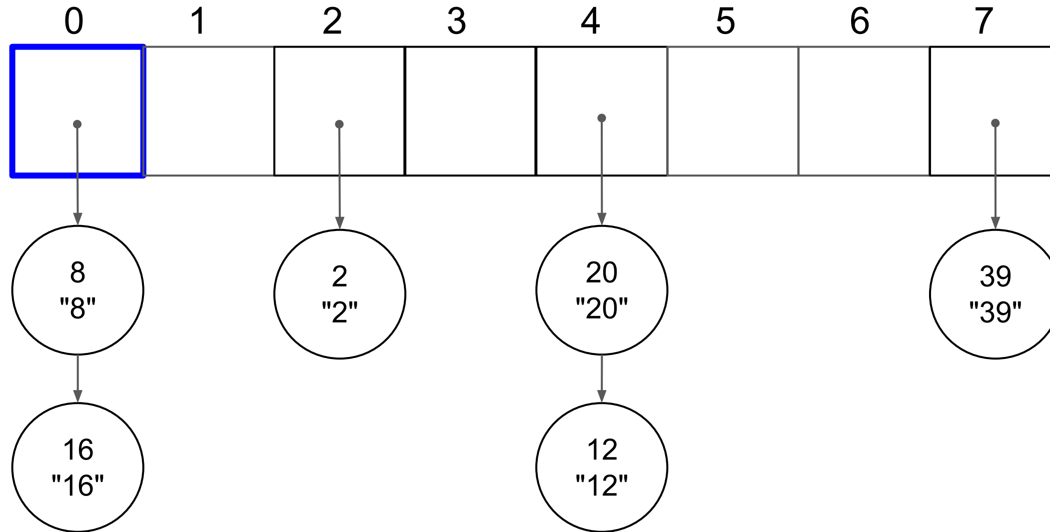


16 % 8 = 0

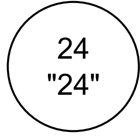


put(16, "16");

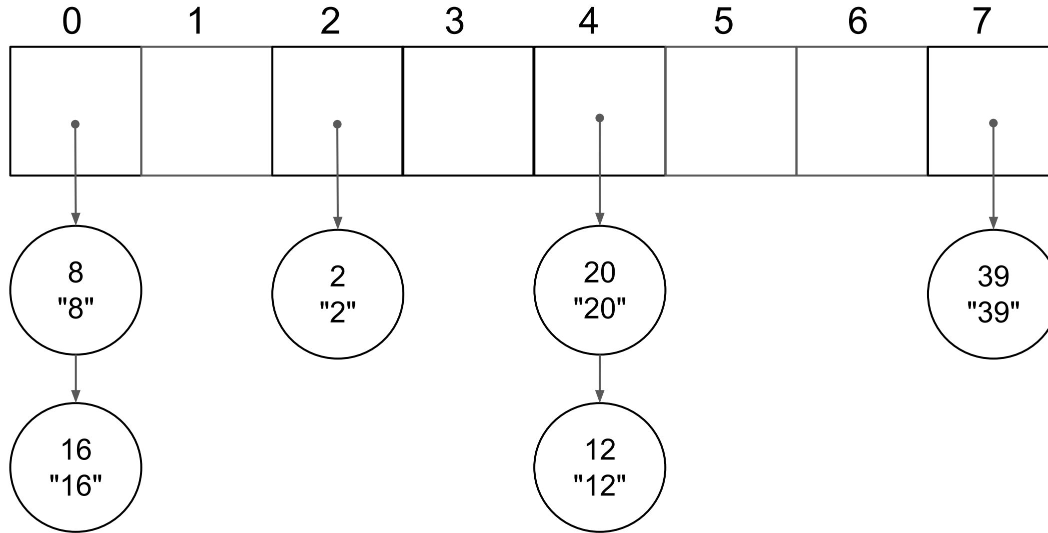
16 % 8 = 0



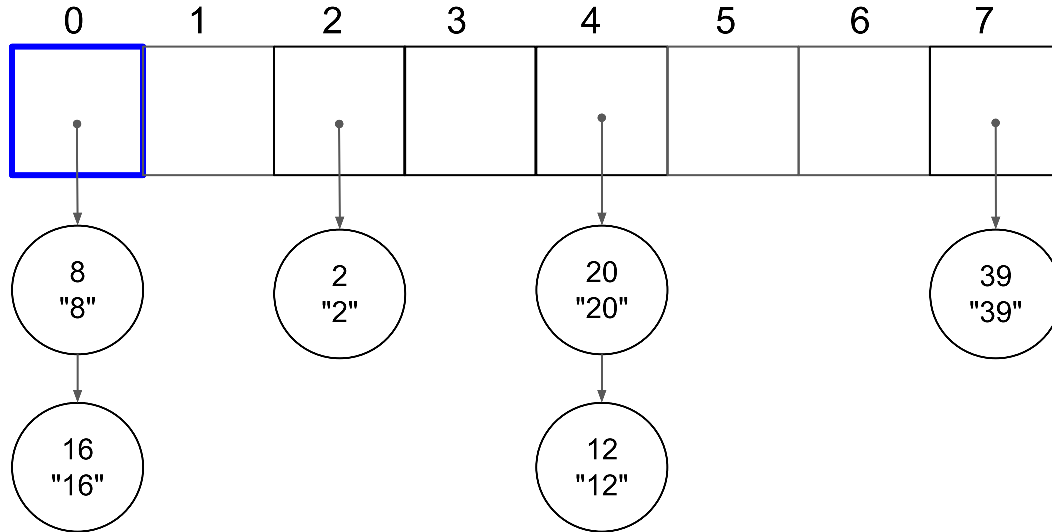
put(24, "24");



24 % 8 = 0

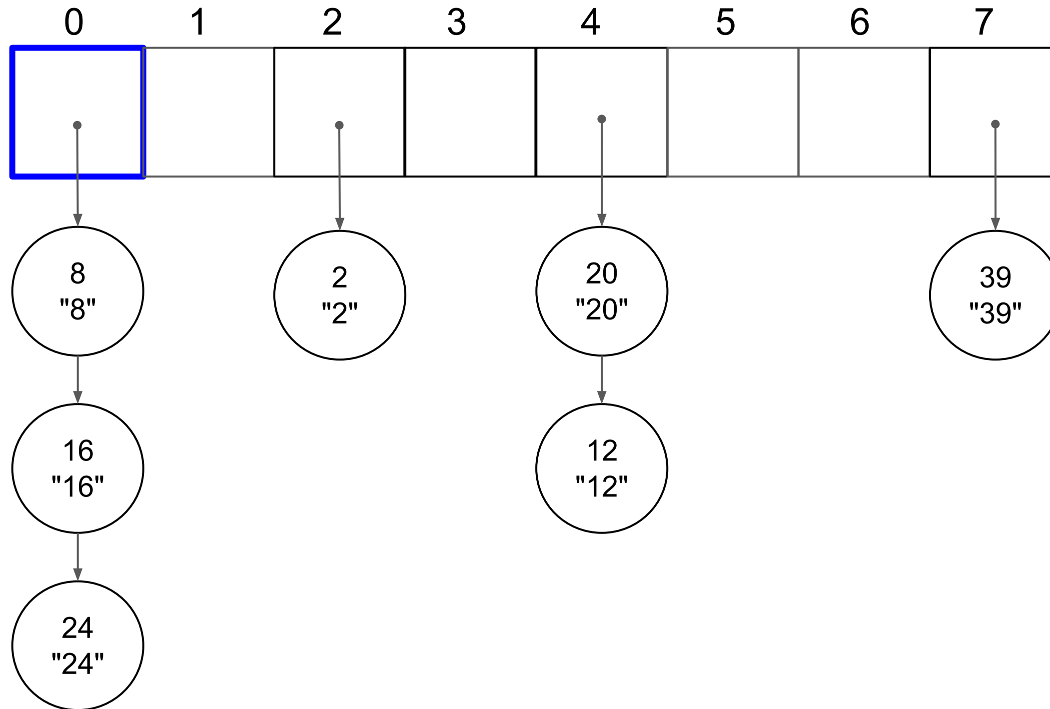


put(24, "24");



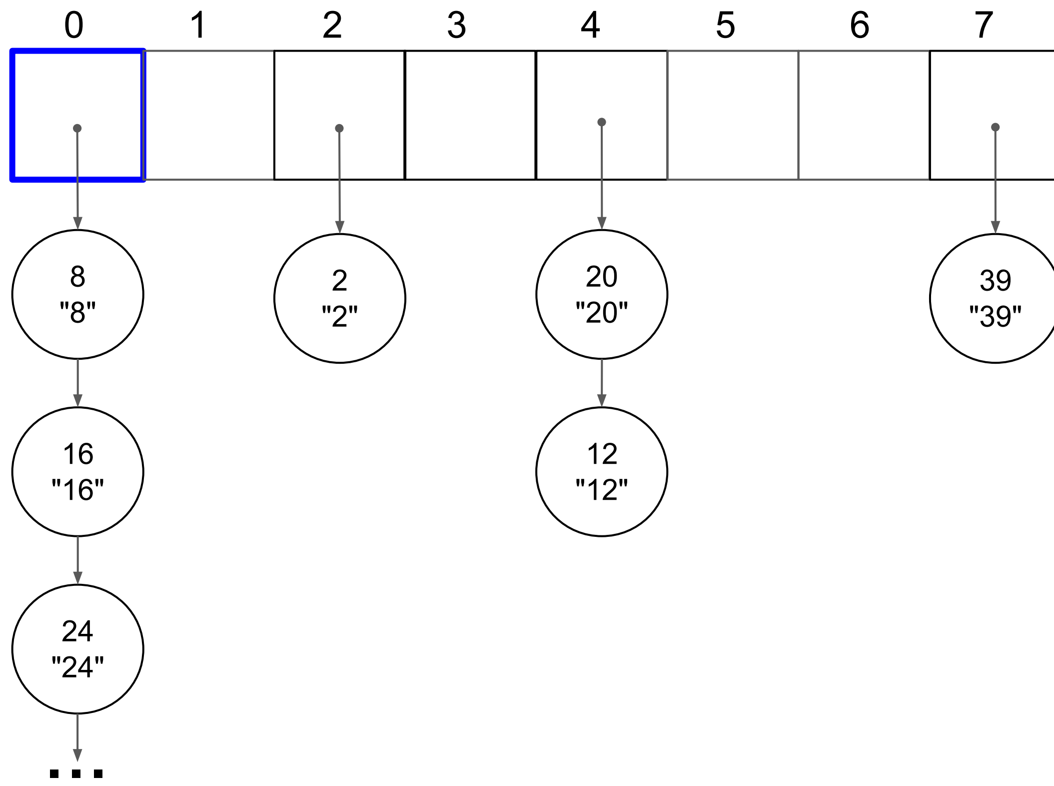
put(24, "24");

24 % 8 = 0

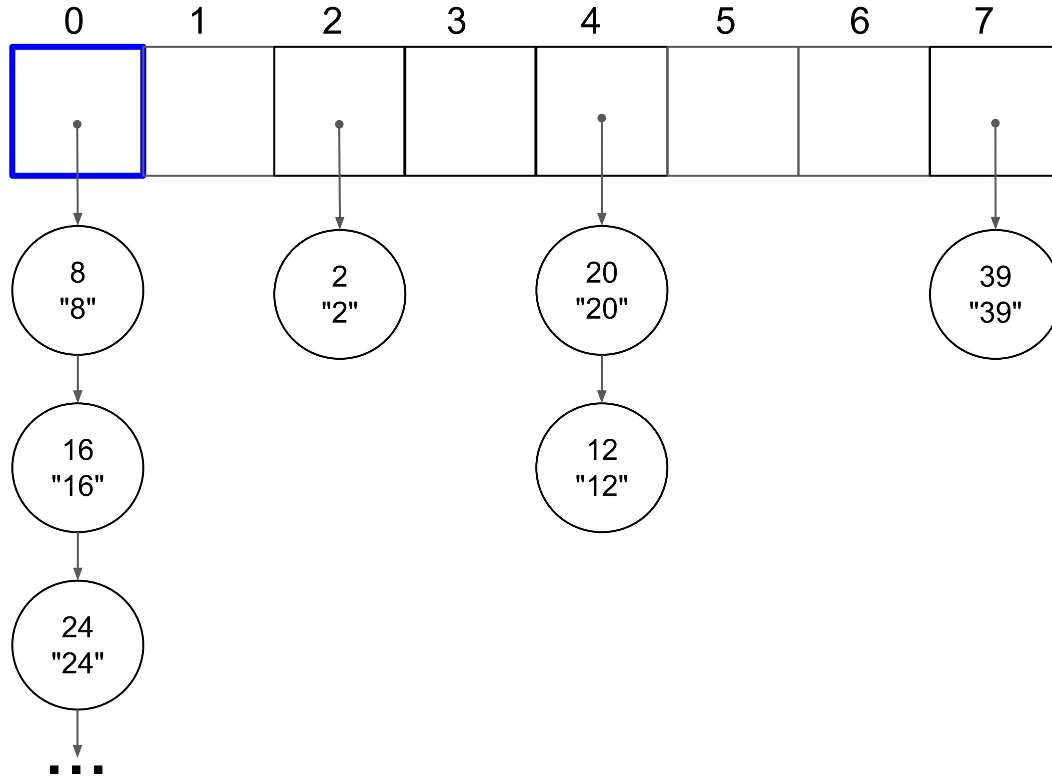


put(32, "32");

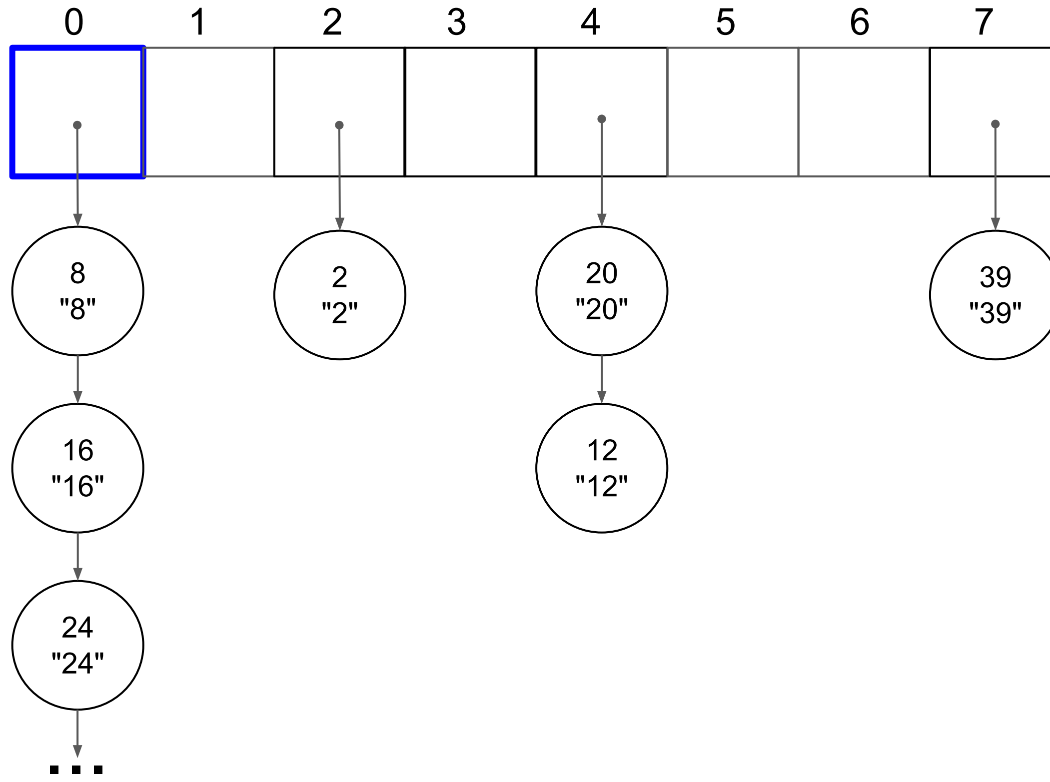
32 % 8 = 0



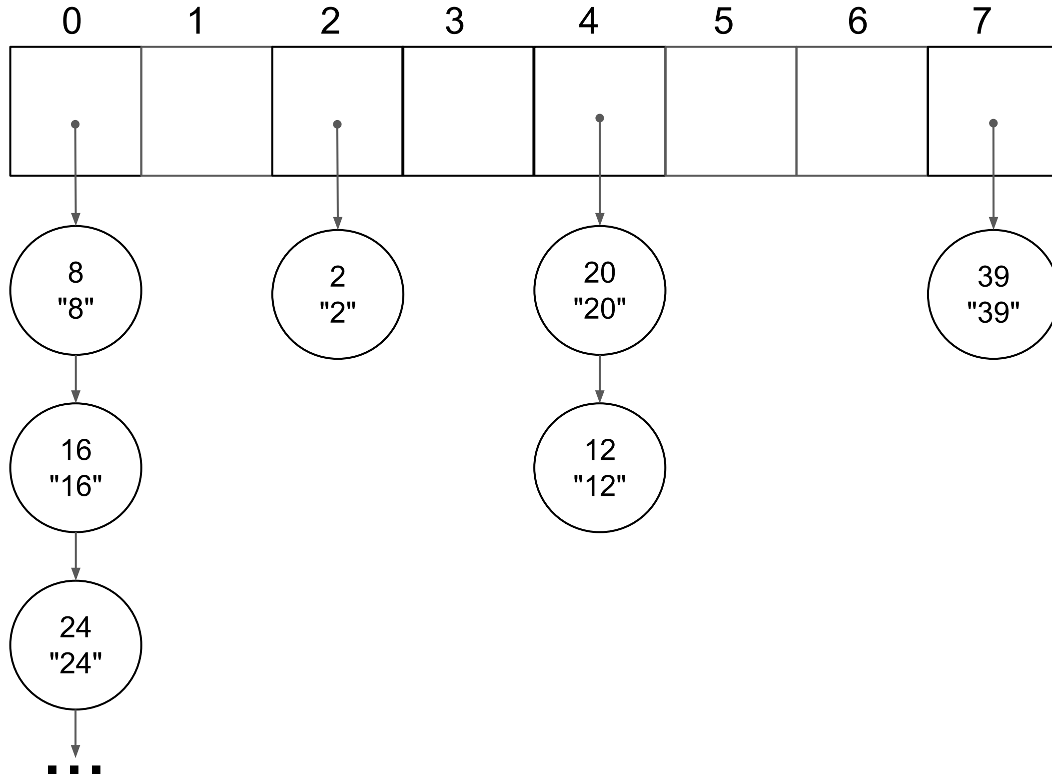
put(40, "40");



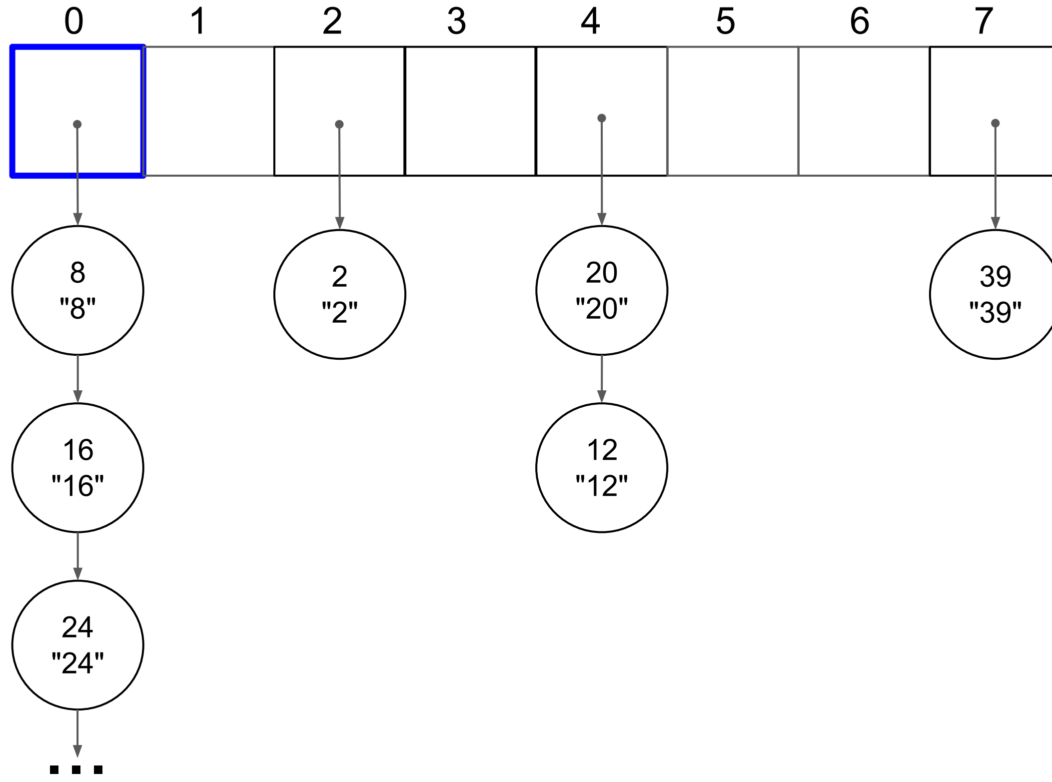
put(48, "48");



get(56);

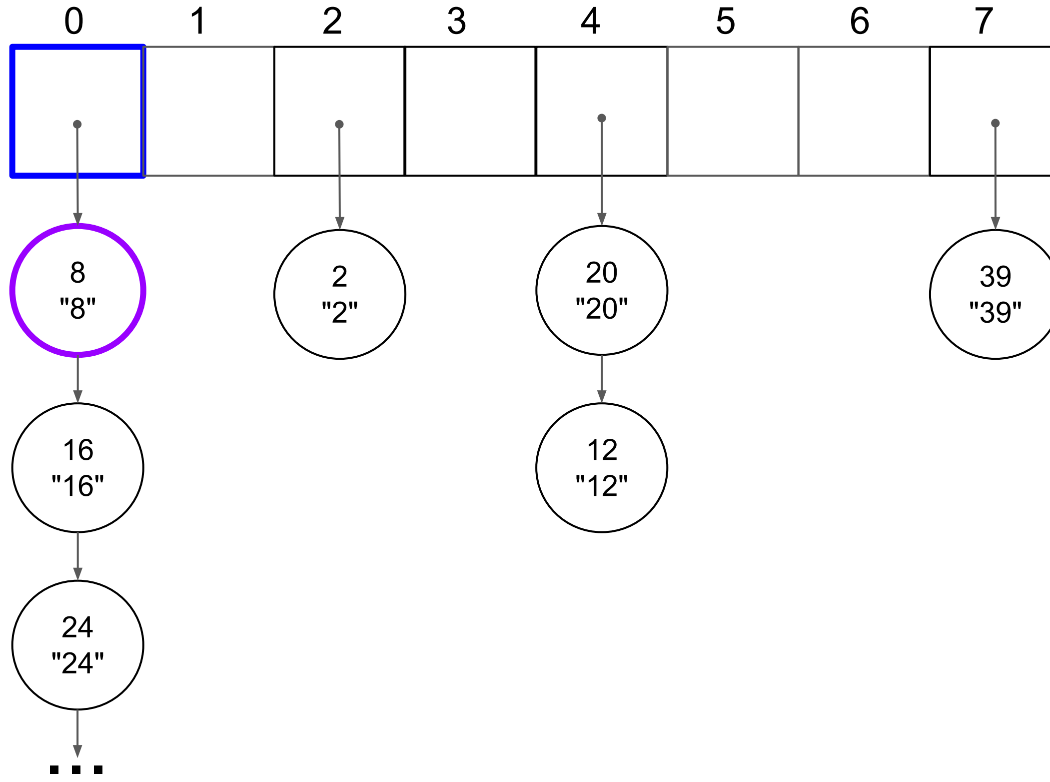


get(56);

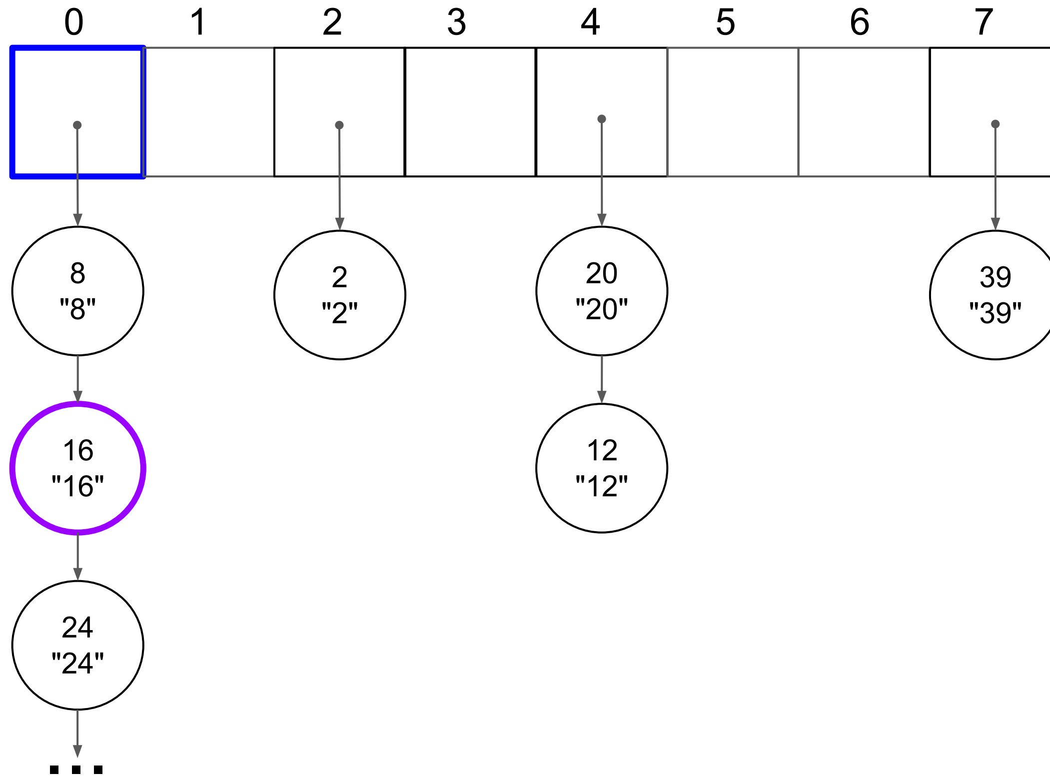


get(56);

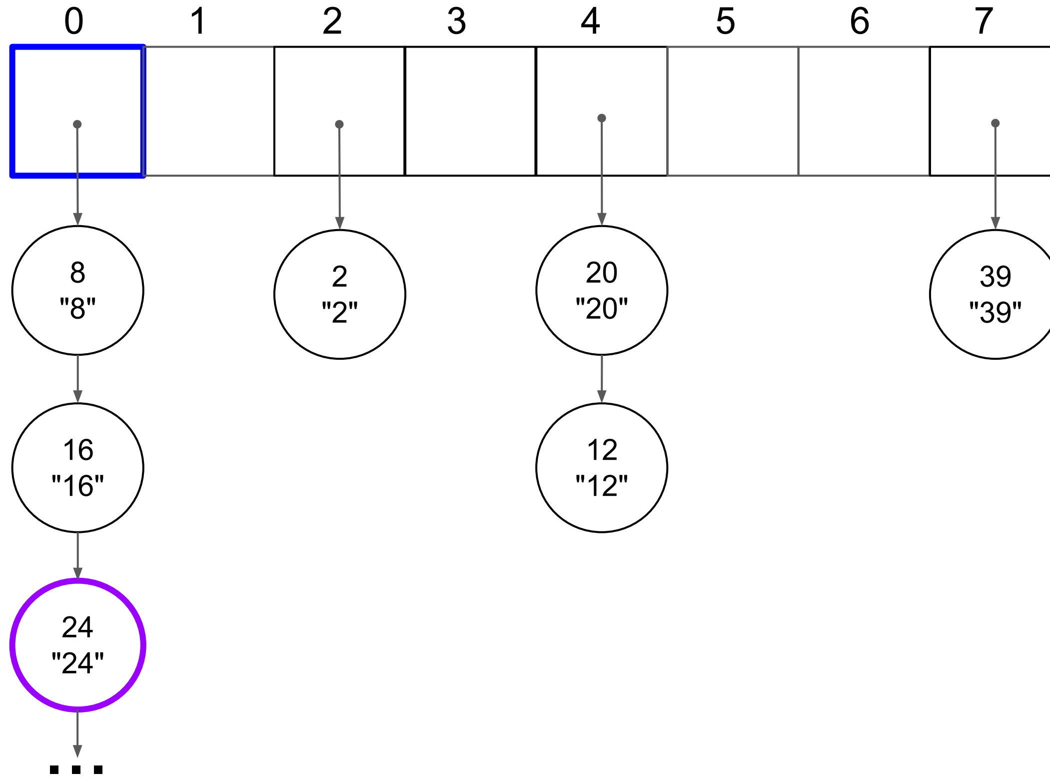
56 % 8 = 0



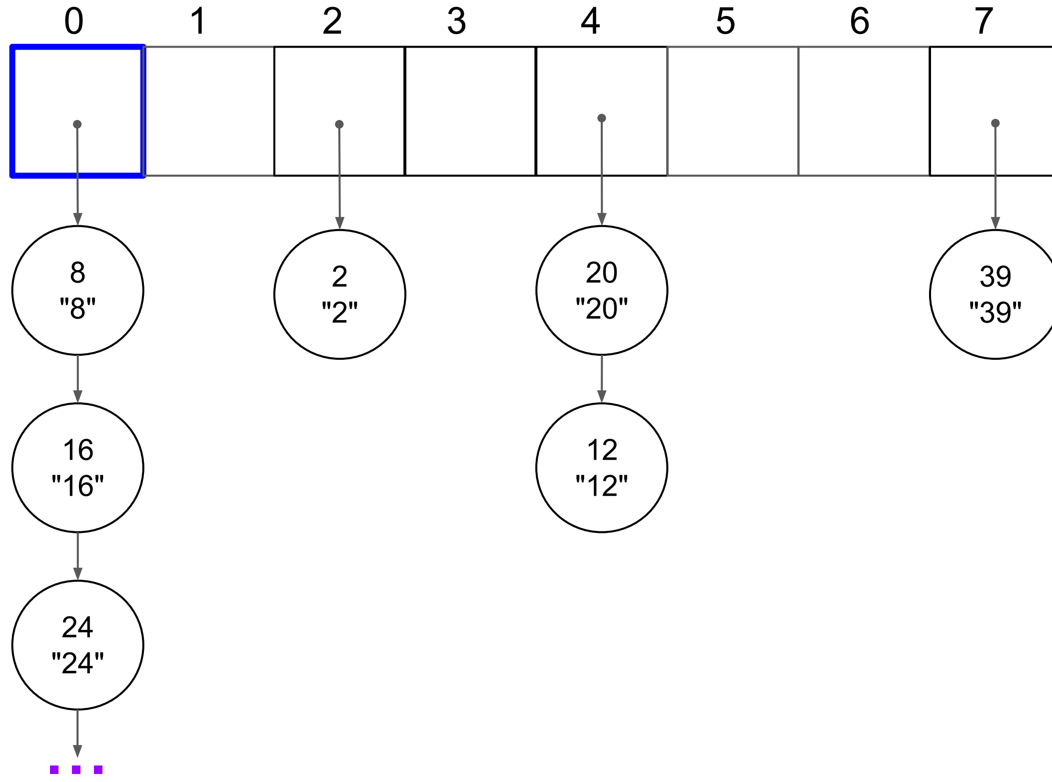
get(56);



get(56);



get(56);



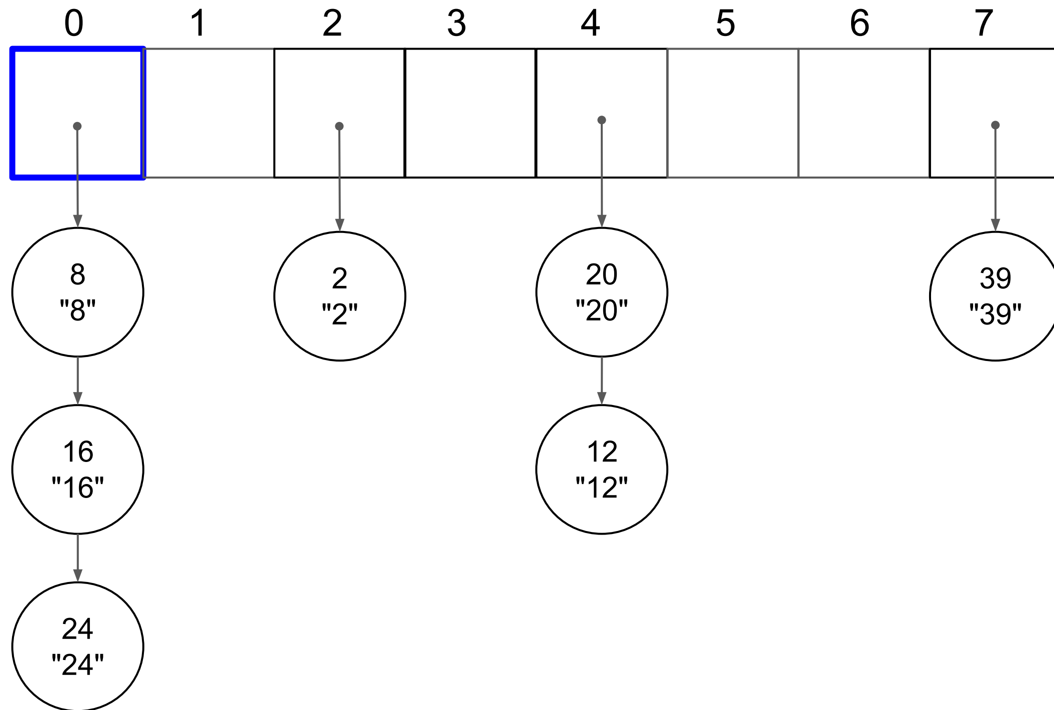
hashing

get(32);

hash code

bucket size (or table size)

32 % 8 = 0



`hashCode = h(key)`

where h is a **hash function**

(Notes from the live demo or live coding. Please do NOT assume the code is complete.)

The hash function that we used in the previous example (which is not really interesting) simply returns the key (positive integer):

```
h(key)
   return key
```

(**compress** operation)

```
index = hashCode % bucketSize
```

What happens if the key is not an integer?

What happens if the key is a string?

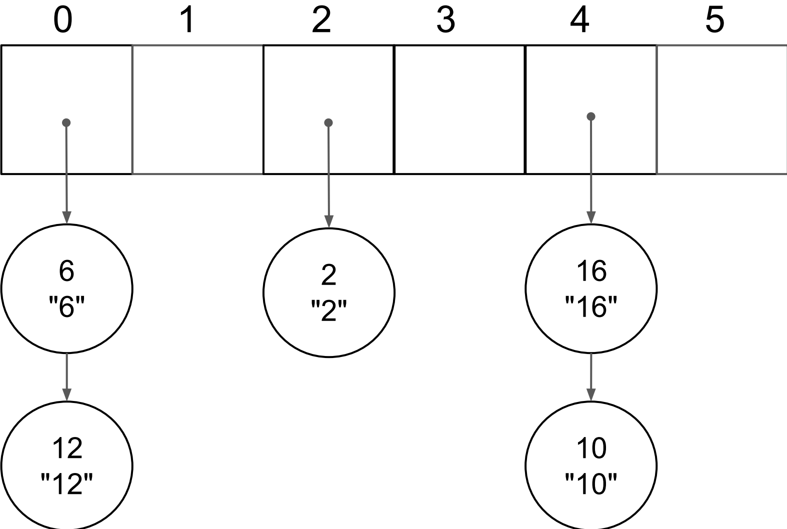
Ascii table

(as an idea)

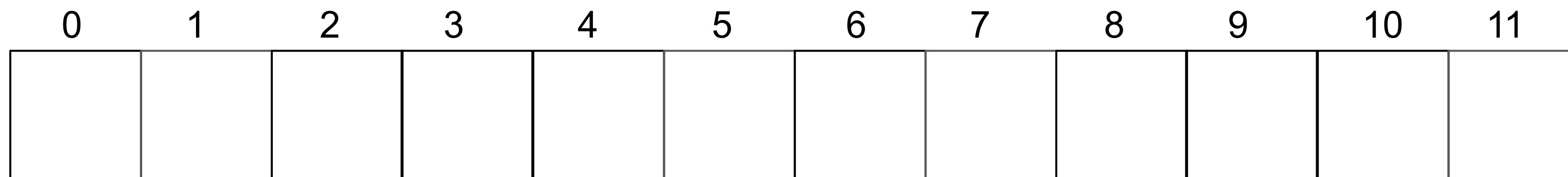
What is a good hash function? (lab8)

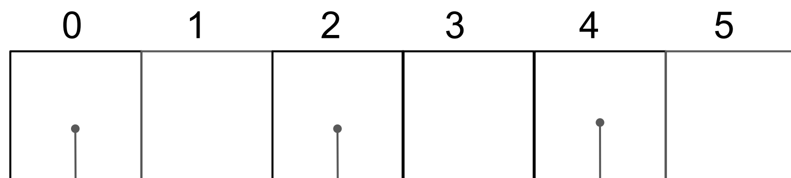
load factor

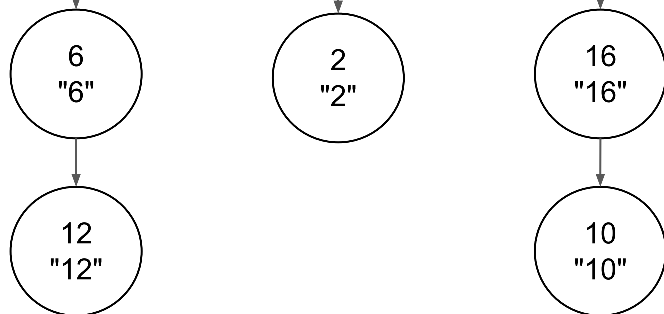
load factor: 5/6



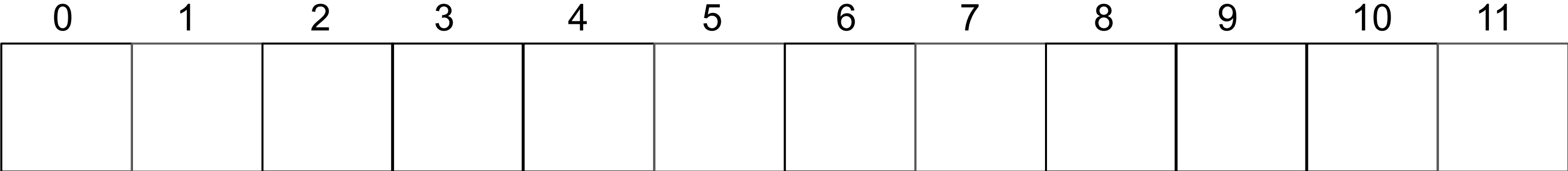
load factor: 5/12

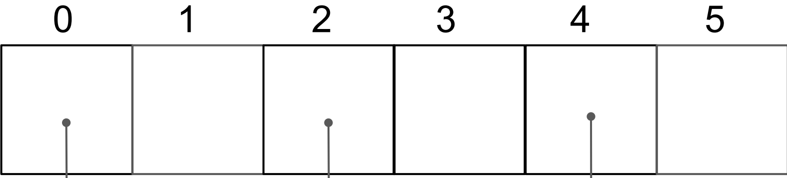


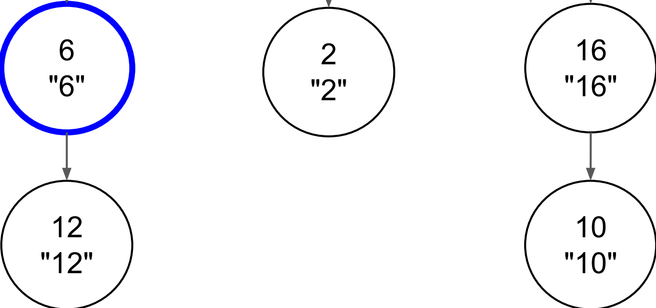




load factor: 5/12

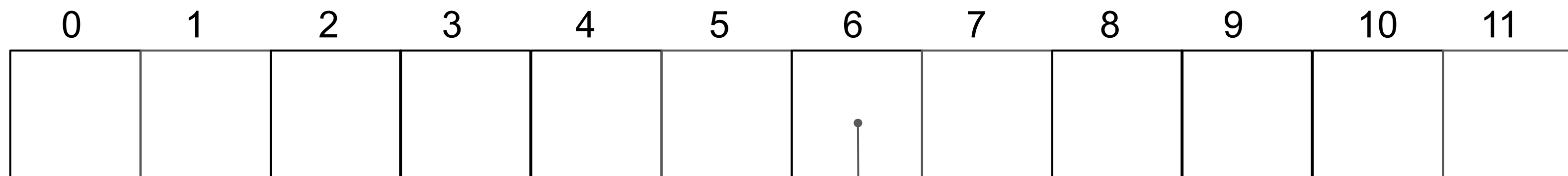


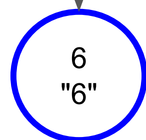


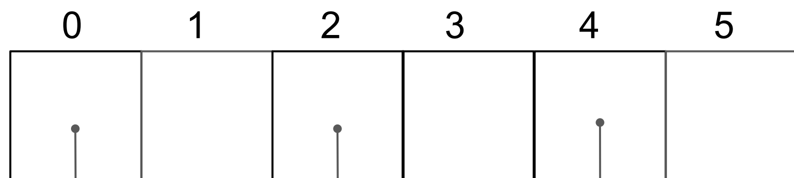


6 % 12 = 6

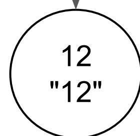
load factor: 5/12

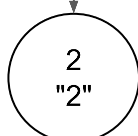


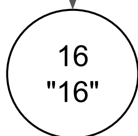


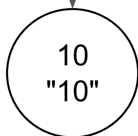


6 % 12 = 6

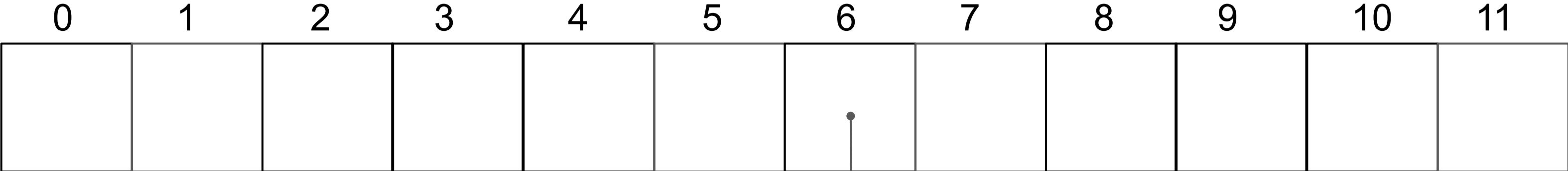


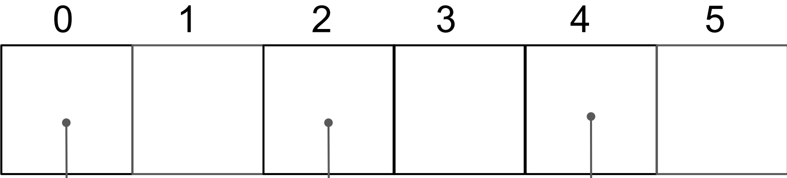






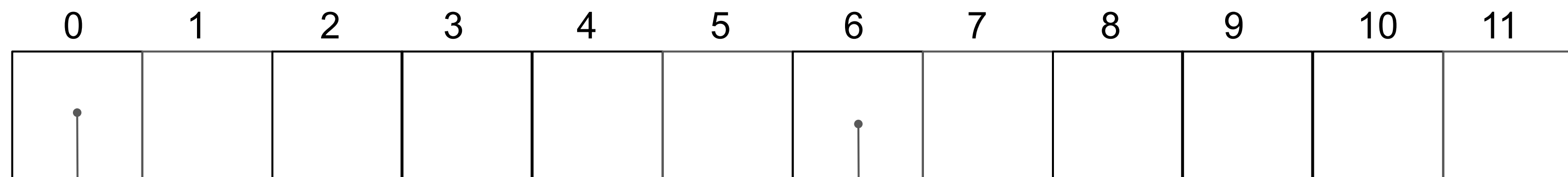
load factor: 5/12

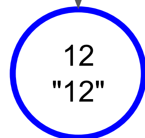


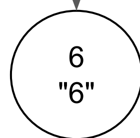


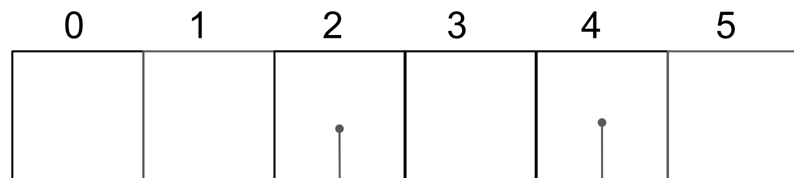
12 % 12 = 0

load factor: 5/12

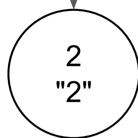


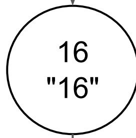


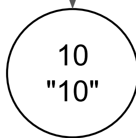




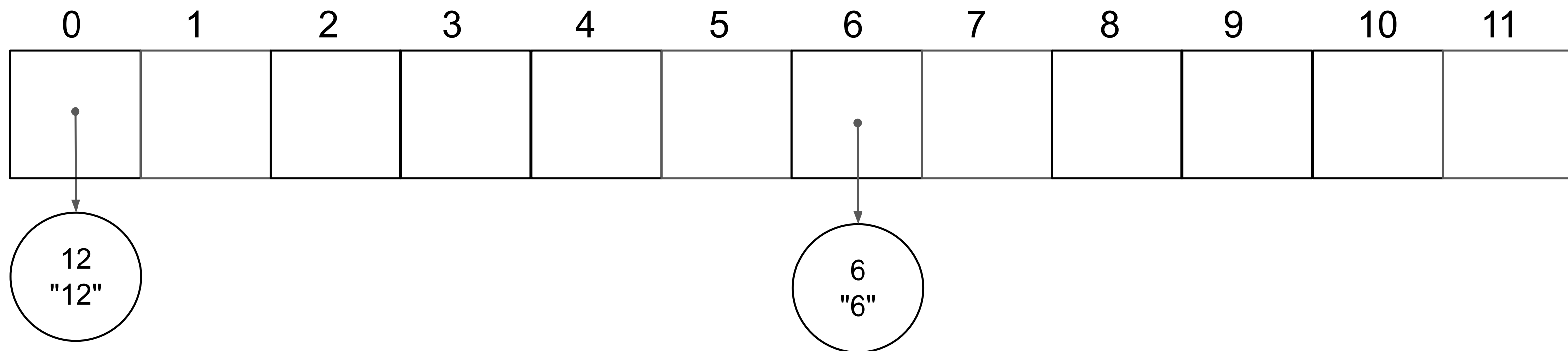
12 % 12 = 0

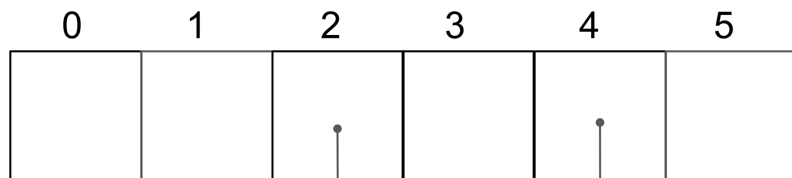


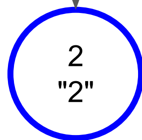


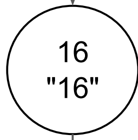


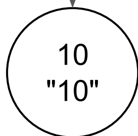
load factor: 5/12





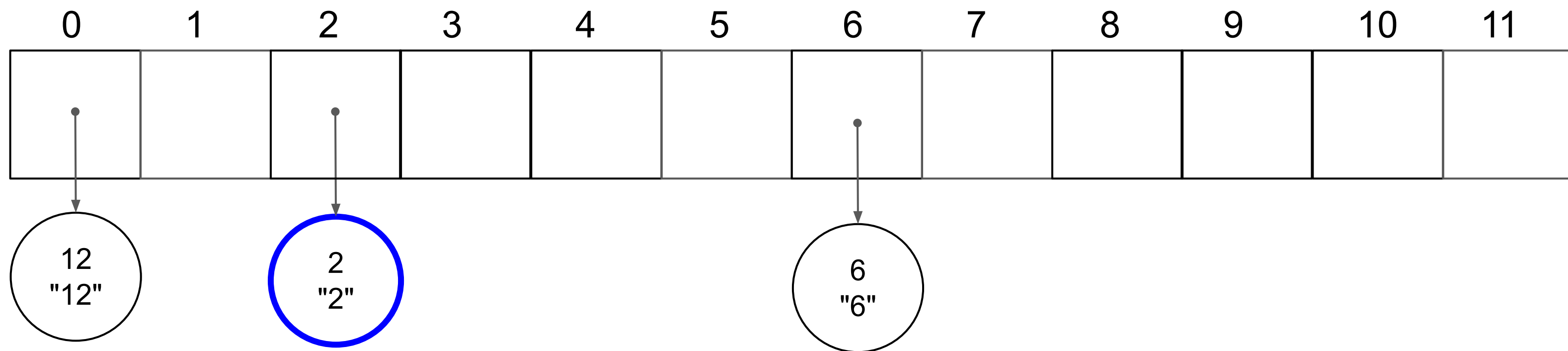


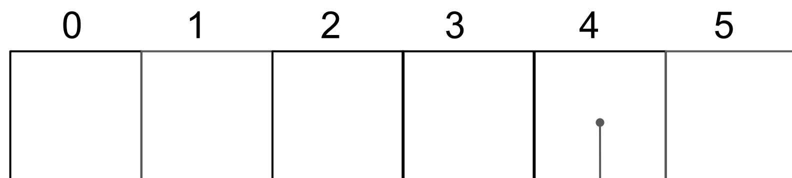


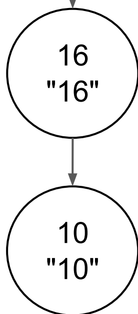


2 % 12 = 2

load factor: 5/12

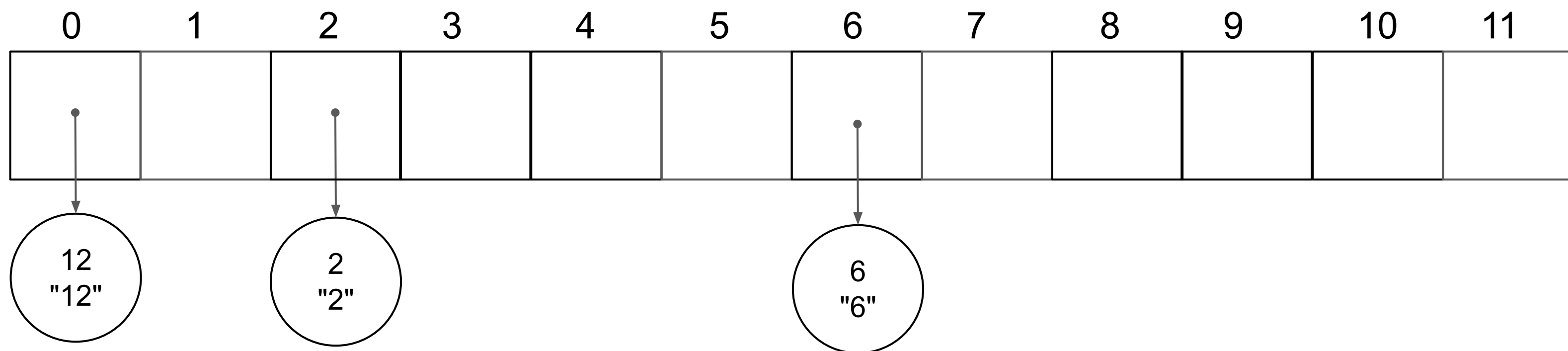


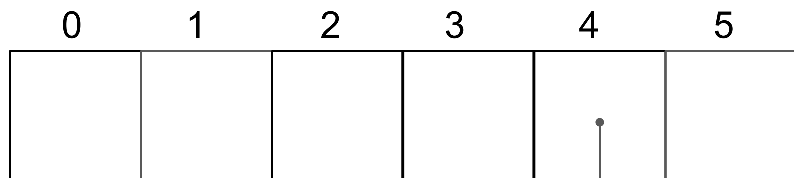


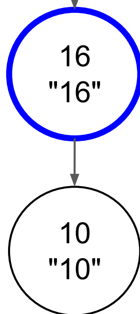


2 % 12 = 2

load factor: 5/12

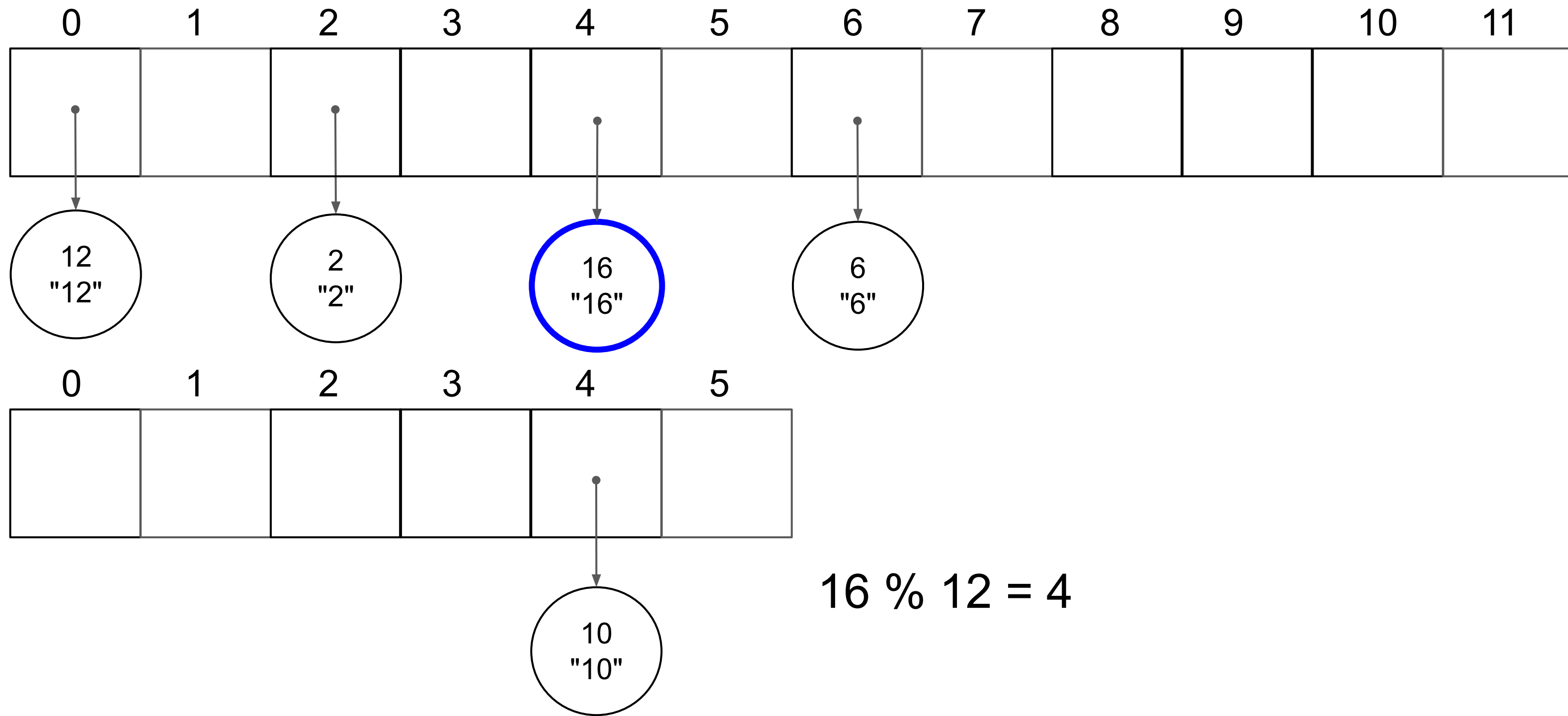






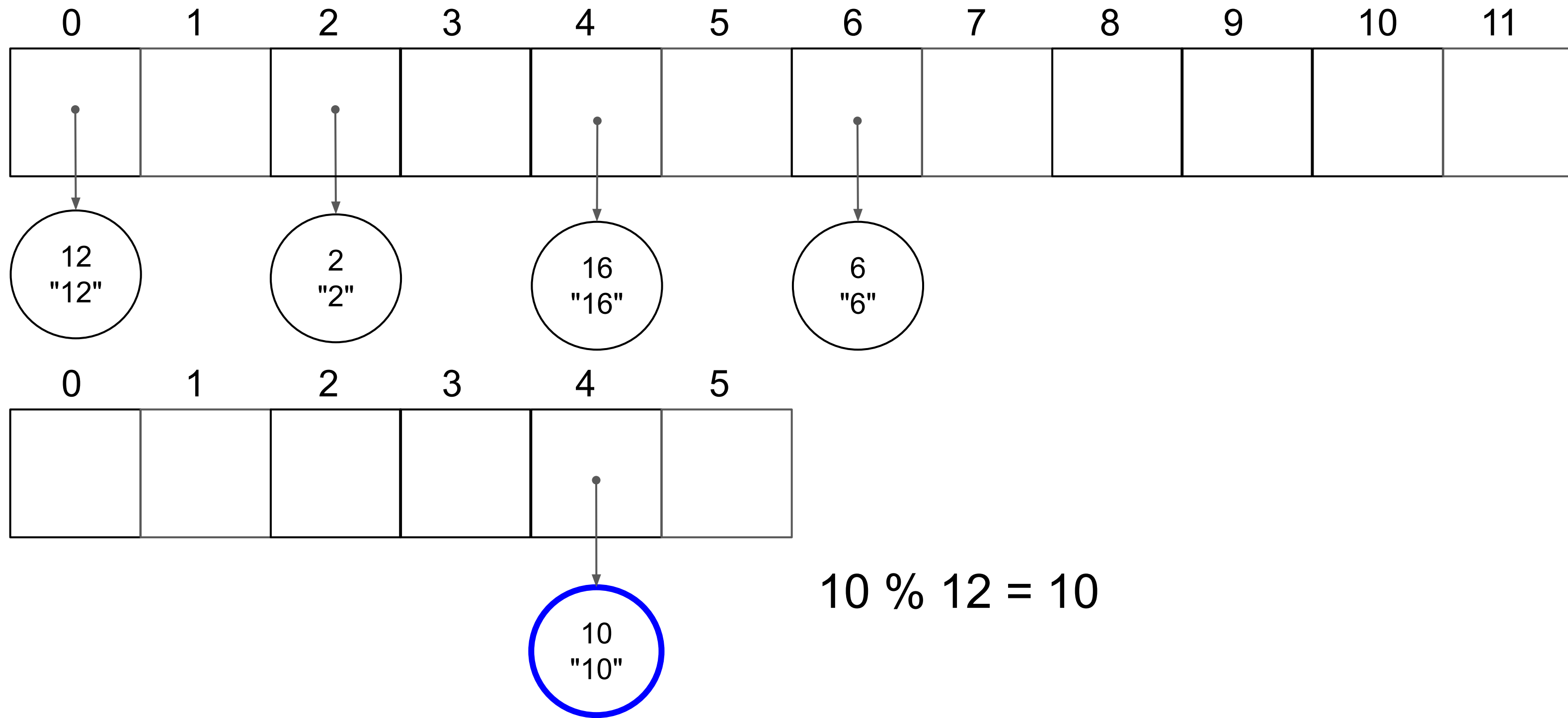
16 % 12 = 4

load factor: 5/12

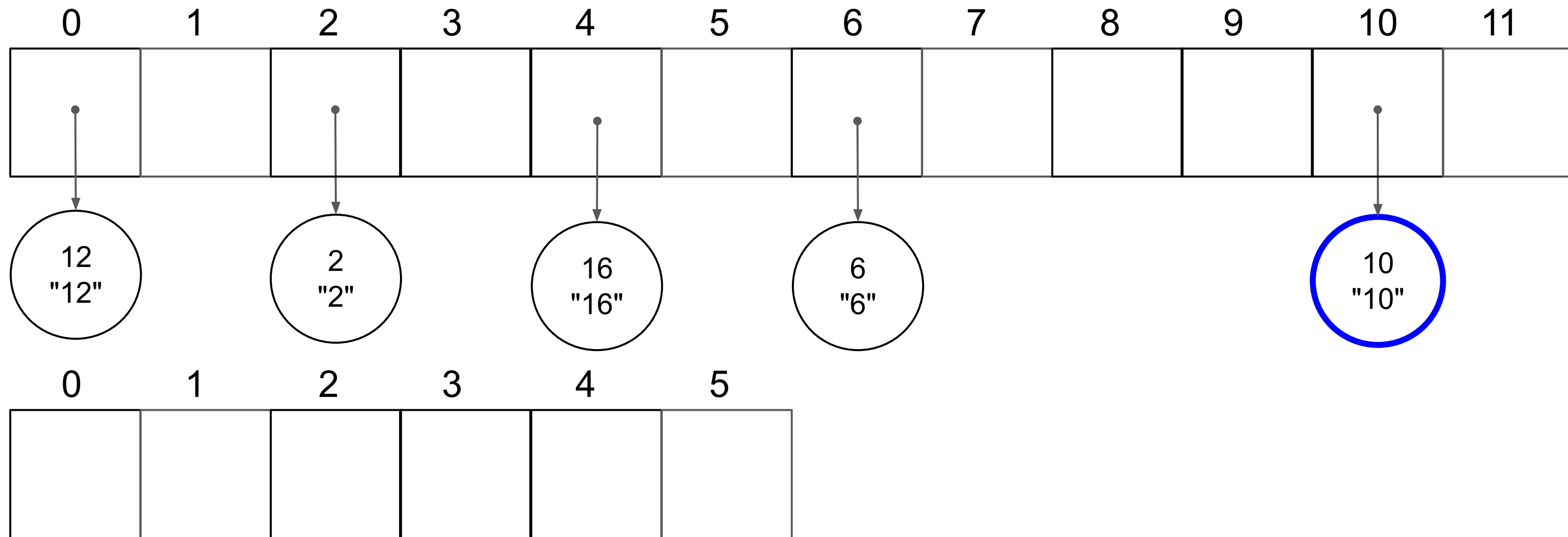


16 % 12 = 4

load factor: 5/12

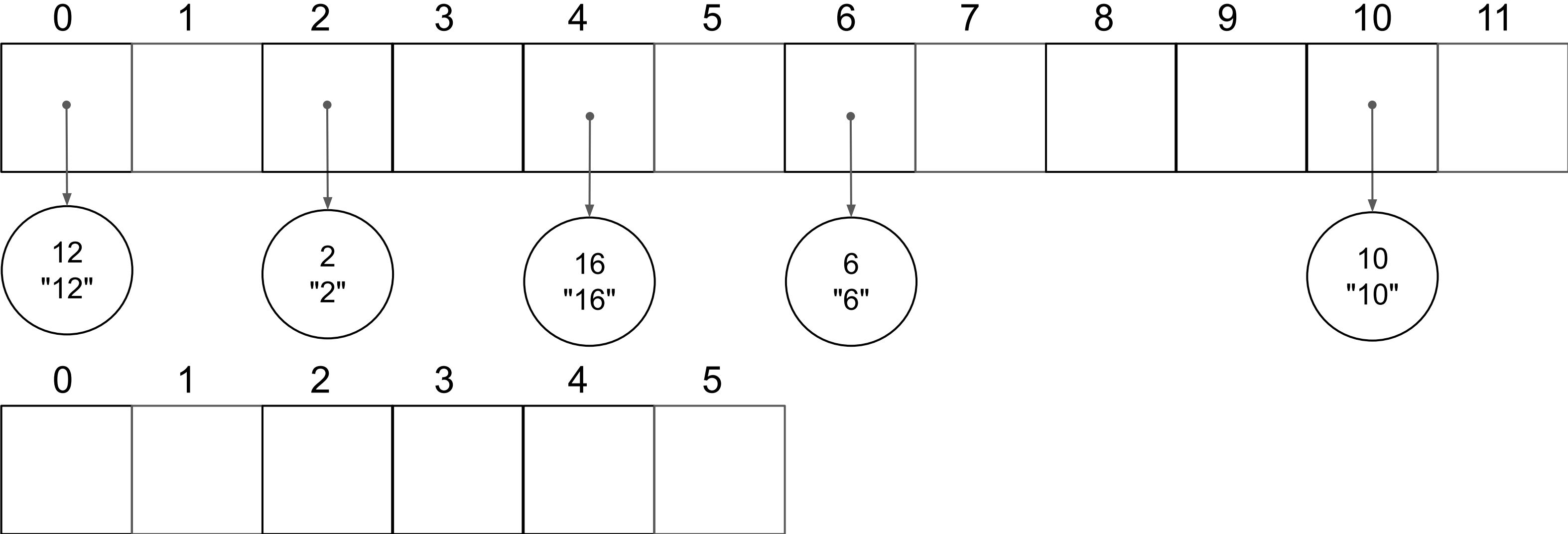


load factor: 5/12

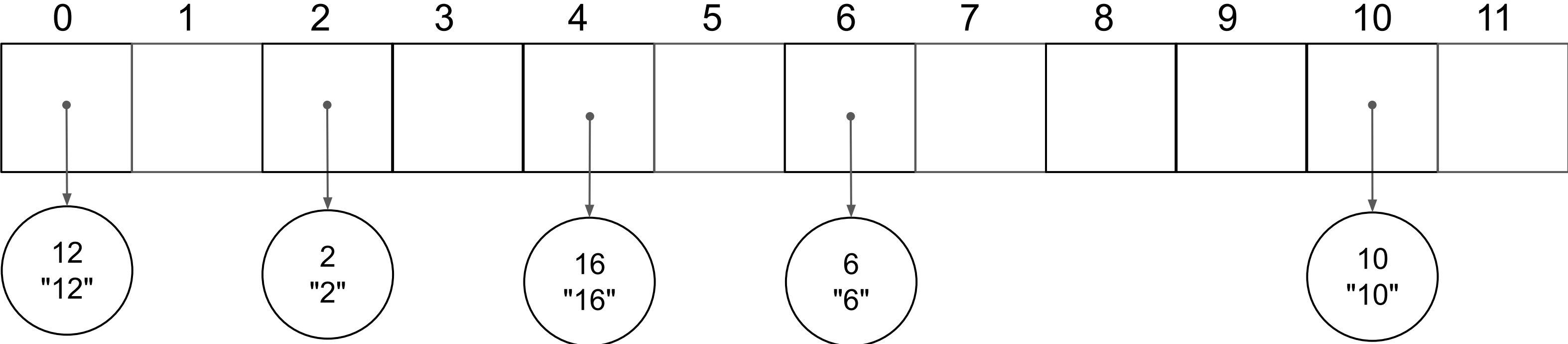


10 % 12 = 10

load factor: 5/12



load factor: 5/12



rehashing

Standard Template Library (STL)

(Notes from the live demo or live coding. Please do NOT assume the code is complete.)

```
 1#include <cassert>
 2#include <iostream>
 3#include <string>
 4#include <vector>
 5
 6int main(){
 7  int n = 20;
 8  std::vector<int> numbers;
 9
10  assert(numbers.empty() == true);
11
12  for(int i = 0; i < n; i++){
13    numbers.push_back(i);
14  }
15
16  for(std::size_t i = 0; i < numbers.size(); i++){
17    //int number = numbers.at(i);
18    int number = numbers[i];
19    std::cout << std::to_string(number) << std::endl;
20  }
21
22  std::vector<std::string> alphabets;
23  alphabets.push_back("A");
24  alphabets.push_back("B");
25  alphabets.push_back("C");
26  alphabets.push_back("D");
27  alphabets.push_back("E");
28  alphabets.push_back("F");
29
30  for(std::vector<std::string>::iterator i = alphabets.begin(); i != alphabets.end(); i++){
31    std::cout << (*i) << std::endl;
32  }
33
34  return 0;
35}
```

Handling collisions

A sneak peek preview (Comp 160, Algorithms)

Open addressing

- Linear probing
- Quadratic probing
- Double hashing

In Your Pocket

arrays
linked lists
stacks
queues
trees
heaps
hash tables

man ssh exit pwd cd ls valgrind touch mkdir cp rm rmdir mv cat head tail less redirect (>, >>, <) pipe (|) (echo, sort, uniq, wc) diff grep clear

Sorting Algorithms
- Selection sort
- Insertion sort
- Merge sort
- Quicksort
- Heapsort
- Counting sort

Some keywords from today's lecture:

- make, Makefile
- Splay tree, zig, zig-zig, zig-zag
- double rotations
- max-heap
- priority queue
- Heapsort (in-place version)
- heap size
- T(n) of binary search
- hash table, put, get, remove, (key-value storage)
- collision
- chaining
- hashing, hash function, hash code, compress, bucket size (table size), rehashing
- ascii table
- load factor
- Standard Template Library (STL), vector, iterator, (stack)

To the lab!