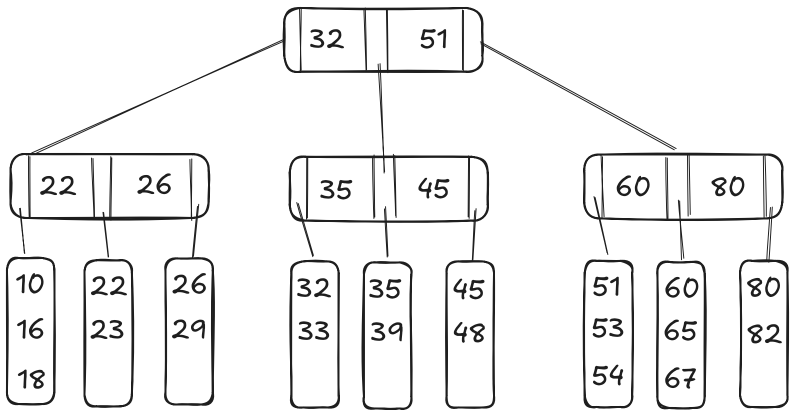
HW 4

Charles Liu
Advanced Data Structures - Dr. Subu Kandaswamy
April 4th, 2025

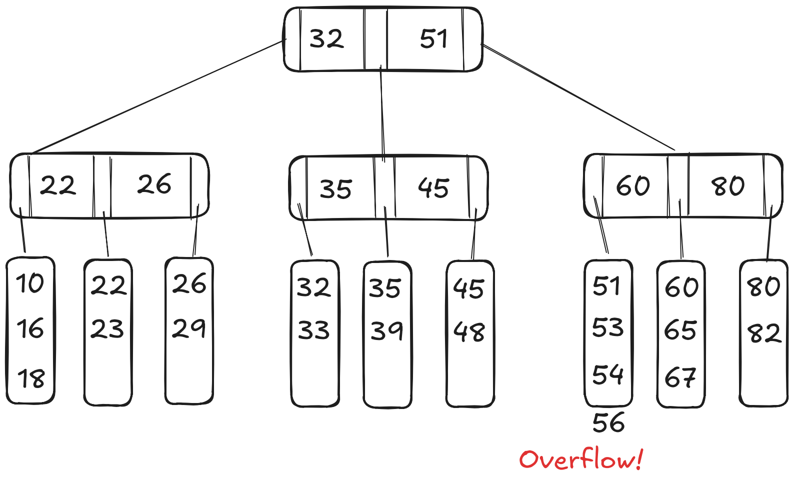
---

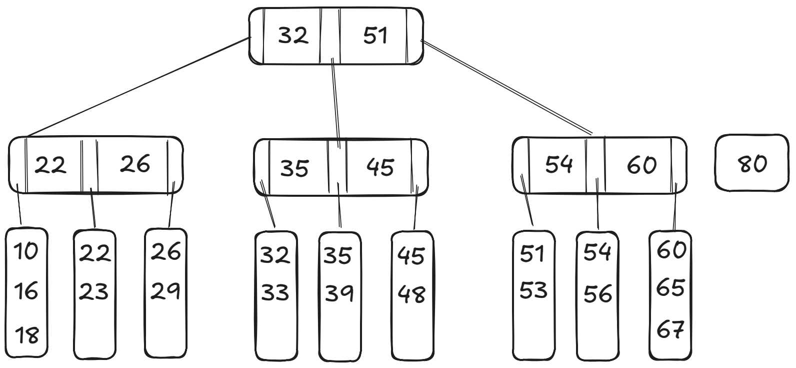
1. (12 points) *B+ tree manipulation*:

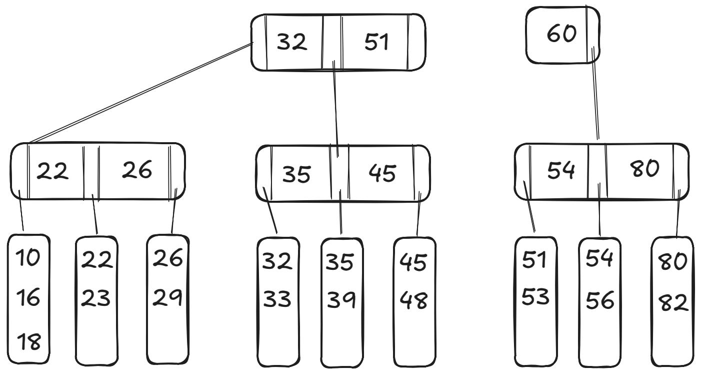
Given the following B+ tree ($M = 3, L = 3$):

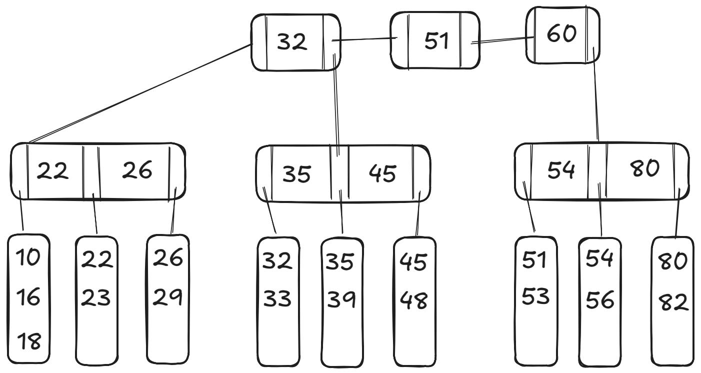


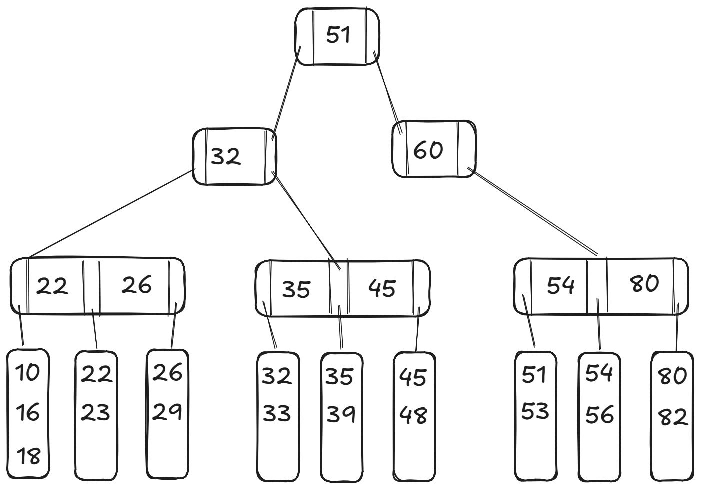
a) Insert data item 56 into the tree and draw the resulting B+ tree. (Note that the height of the tree will increase after the insertion.)



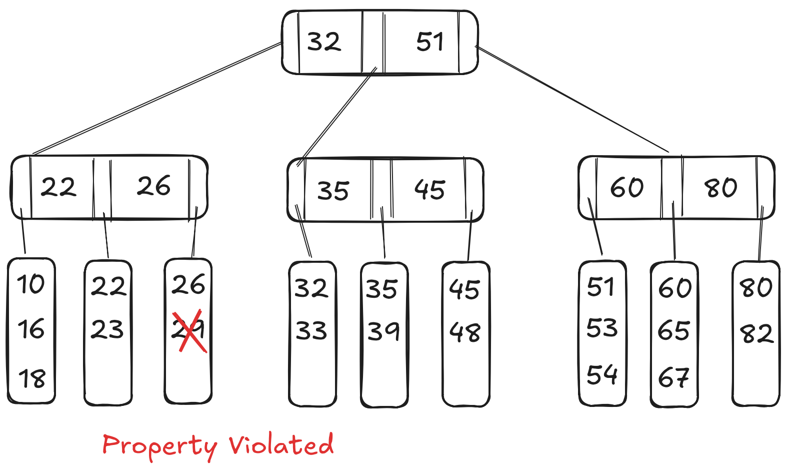

32 51
22 26
35 45
54 60
80
10 16 18
22 23
26 29
32 33
35 39
45 48
51 53
54 56
60 65 67

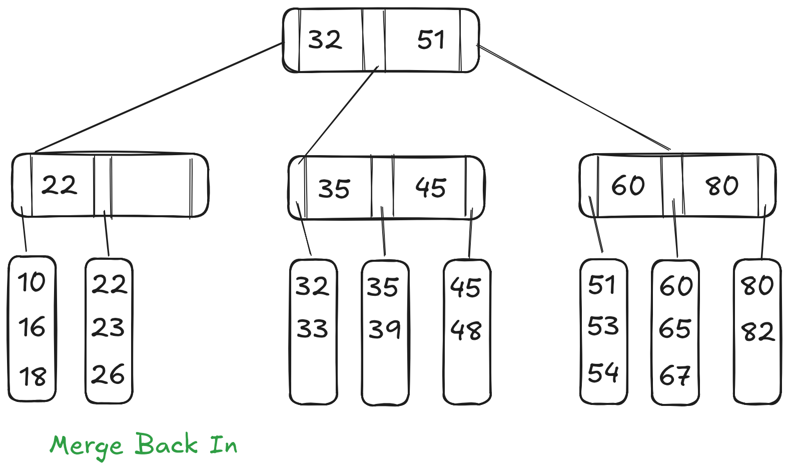

32 51
60
22 26
35 45
54 80
10 16 18
22 23
26 29
32 33
35 39
45 48
51 53
54 56
80 82


32
51
60
22 26
35 45
54 80
10 16 18
22 23
26 29
32 33
35 39
45 48
51 53
54 56
80 82


51
32
60
22 26
35 45
54 80
10 16 18
22 23
26 29
32 33
35 39
45 48
51 53
54 56
80 82

b) Delete data item 29 in the original tree (the above figure) and draw the resulting B+ tree. (The height of the tree should not change after the deletion.)





2. (4 points) *Hash table sizes:*

If you wanted to implement a hash table, then which of these would probably be the best initial table size to pick?

Table size choices:

$$7 \quad 100 \quad 101 \quad 27 \quad 525$$

Why did you choose that one? Briefly explain the rationale.
I would pick 101. It is prime so you prevent endless or highly inefficiency cycling with probing.

3. (16 points) *Hash table collision resolution:*

For each of the different hash tables described below (parts [a] through [d]), show the final hash table after inserting the following keys (in order) into an initially empty table with no rehashing:

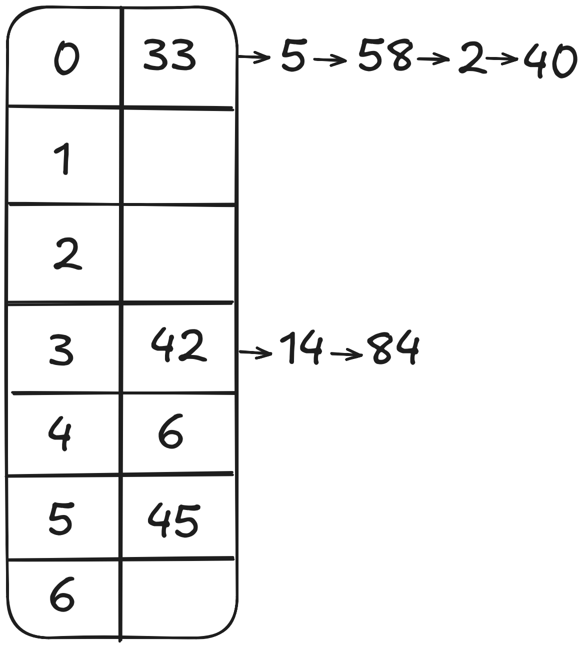
$\{42, 33, 45, 5, 14, 58, 84, 6, 2, 40\}$

Note: Your solutions should match the result from using the code in the textbook, which may be different from the results obtained by web apps or other implementations.

a) A hash table of size $M = 7$ using collision-resolution by chaining and the hash function:

$$hash(x) = (x \times x + 3) \bmod M$$

$hash(42) = (42 \times 42 + 3) \bmod M = 3$
$hash(33) = (33 \times 33 + 3) \bmod M = 0$
$hash(45) = (45 \times 45 + 3) \bmod M = 5$
$hash(5) = (5 \times 5 + 3) \bmod M = 0$
$hash(14) = (14 \times 14 + 3) \bmod M = 3$
$hash(58) = (58 \times 58 + 3) \bmod M = 0$
$hash(84) = (84 \times 84 + 3) \bmod M = 3$
$hash(6) = (6 \times 6 + 3) \bmod M = 4$
$hash(2) = (2 \times 2 + 3) \bmod M = 0$
$hash(40) = (40 \times 40 + 3) \bmod M = 0$



b) A hash table of size M=11 using collision-resolution by open-addressing and the linear probing hash function:

$$h_i(x) = (hash(x) + f(i)) \bmod M$$

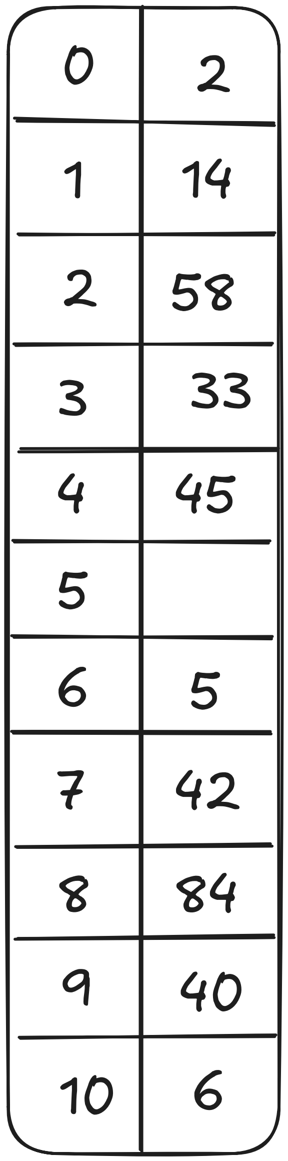
where: $hash(x) = (x \times x + 3) \bmod M$, and $f(i) = i$.

0	40
1	14
2	58
3	33
4	45
5	
6	5
7	42
8	84
9	6
10	2

c) A hash table of size M=11 using collision-resolution by open-addressing and the quadratic probing hash function:

$$h_i(x) = (hash(x) + f(i)) \bmod M$$

where: $hash(x) = (x \times x + 3) \bmod M$, and $f(i) = i^2$.

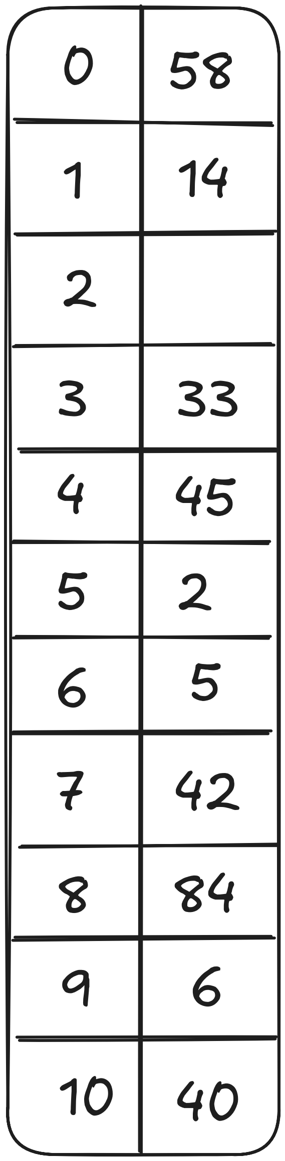


0	2
1	14
2	58
3	33
4	45
5	
6	5
7	42
8	84
9	40
10	6

d) A hash table of size M=11 using collision-resolution by open-addressing and the double hashing function probing hash function:

$$h_i(x) = (hash(x) + f(i)) \bmod M$$

where: $hash(x) = (x \times x + 3) \bmod M$, and $f(i) = i \times hash_2(x)$ and $hash_2(x) = R – (x \bmod R)$, where $R = 7$.



0	58
1	14
2	
3	33
4	45
5	2
6	5
7	42
8	84
9	6
10	40

4. (6 points) Hash table insertion:

The hash table below works, but once we put more than a few thousand entries, the whole thing starts to slow down. Searches, inserts, and contains calls start taking *much* longer than $O(1)$ time. It is problematic because it is slowing down the whole application services backend. We think the performance issue stems from the rehash code, but we are not sure where. State a reason why you would think the hash table starts to perform poorly as it grows bigger. You can point to the part(s) of the code that you think are potentially causing the problem and discuss why. Then, propose a modification that is likely to fix that problem.

```
/**
* Rehashing for linear probing hash table
```

```
*/

void rehash() {
    vector<HashEntry> oldArray = array;
    //create new double-sized, empty table
    array.resize(2 * oldArray.size());

    for (auto &entry: array) {
        entry.info = EMPTY;
    }

    //copy table over
    currentSize = 0;
    for (auto &entry : oldArray) {
        if (entry.info == ACTIVE) {
            insert(std::move(entry.element));
        }
    }
}
```

Consider the line of code `array.resize(2 * oldArray.size());`

The rehash sizes the array to 2 times the old array. This makes it a composite number, which increases likelihood of different probing methods such as quadratic probing getting lost in cycles, greatly decreasing performance.

One modification is you could do is double the array size, go to the table of all known primes, and find the next largest prime after that number.