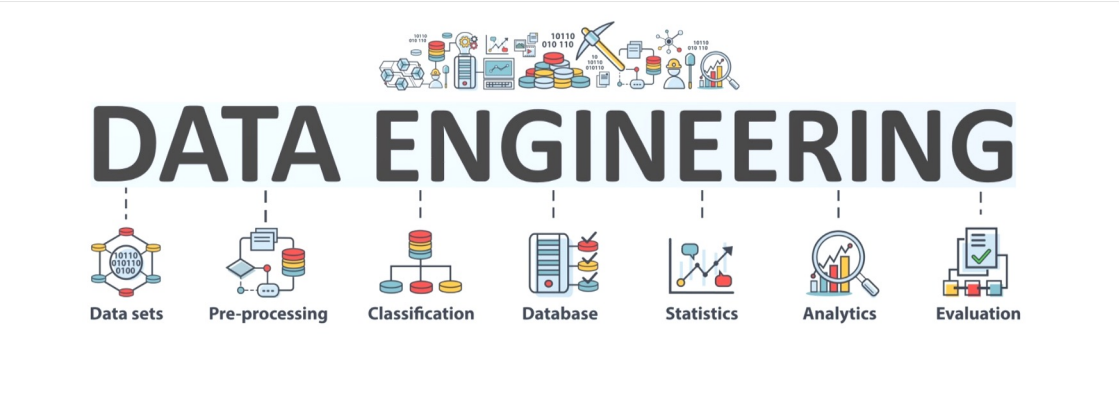


Lecture 12:

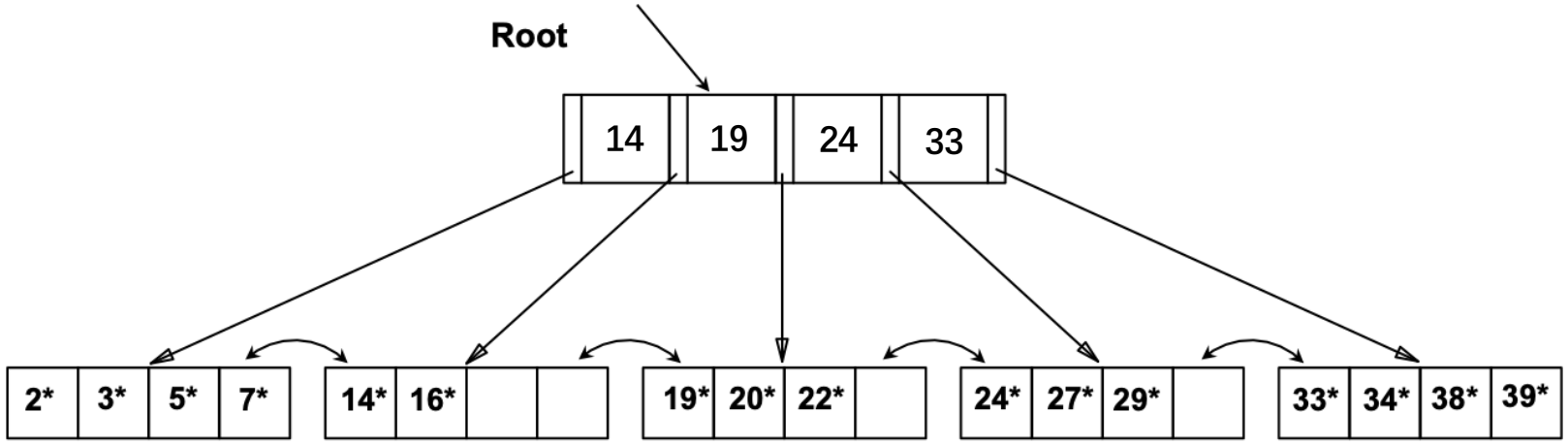
Exercises

CS5481 Data Engineering

Instructor: Yifan Zhang

Exercise1: B+ Tree

- **Search begins at root, and key comparisons direct it to a leaf**
- **Search for 5*, 15*, all data entries >= 24* ...**



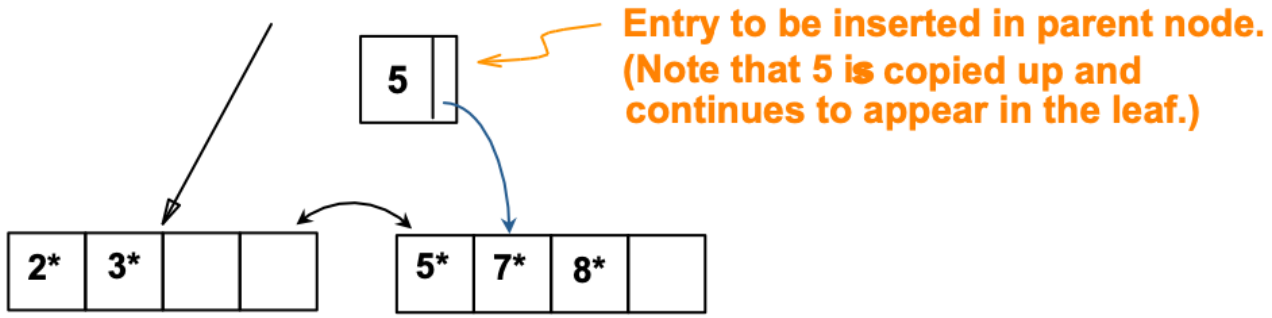
☛ *Based on the search for 15*, we know it is not in the tree!*

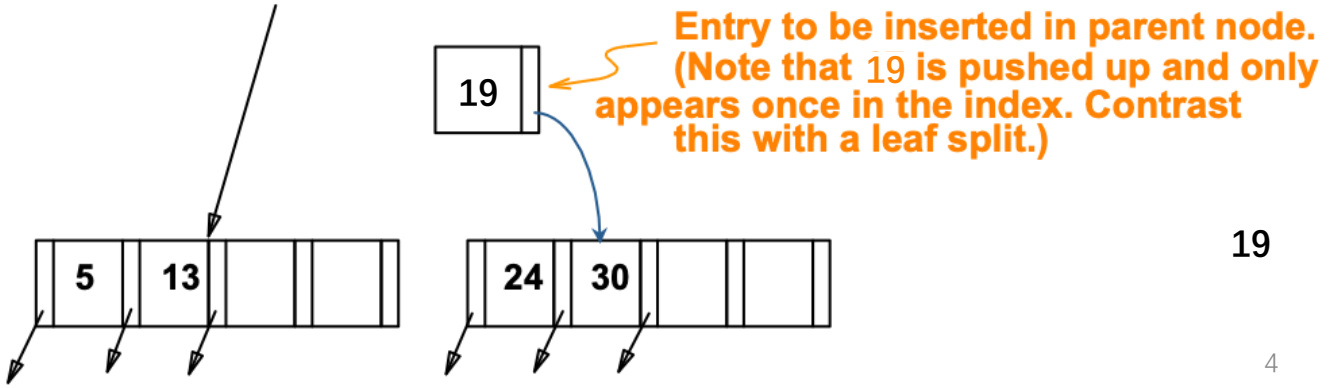
Review: B+ Tree Insertion

- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Exercise1: Insert 8* into Example B+ Tree

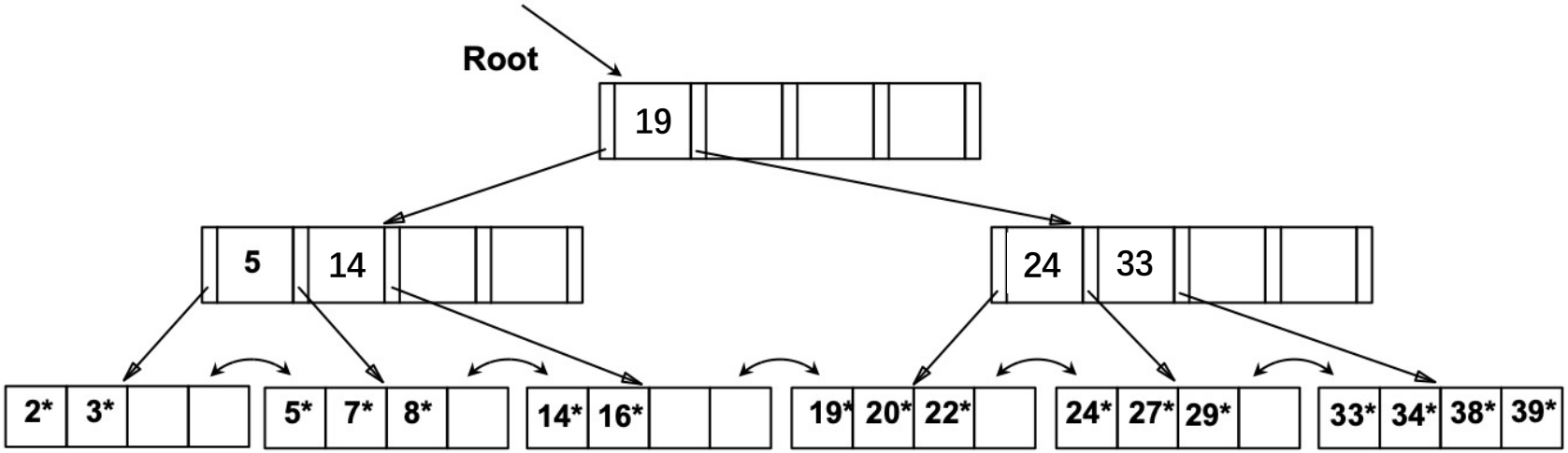
- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up;* be sure you understand the reasons for this.





19

Exercise1: Inserting 8* into Example B+ Tree

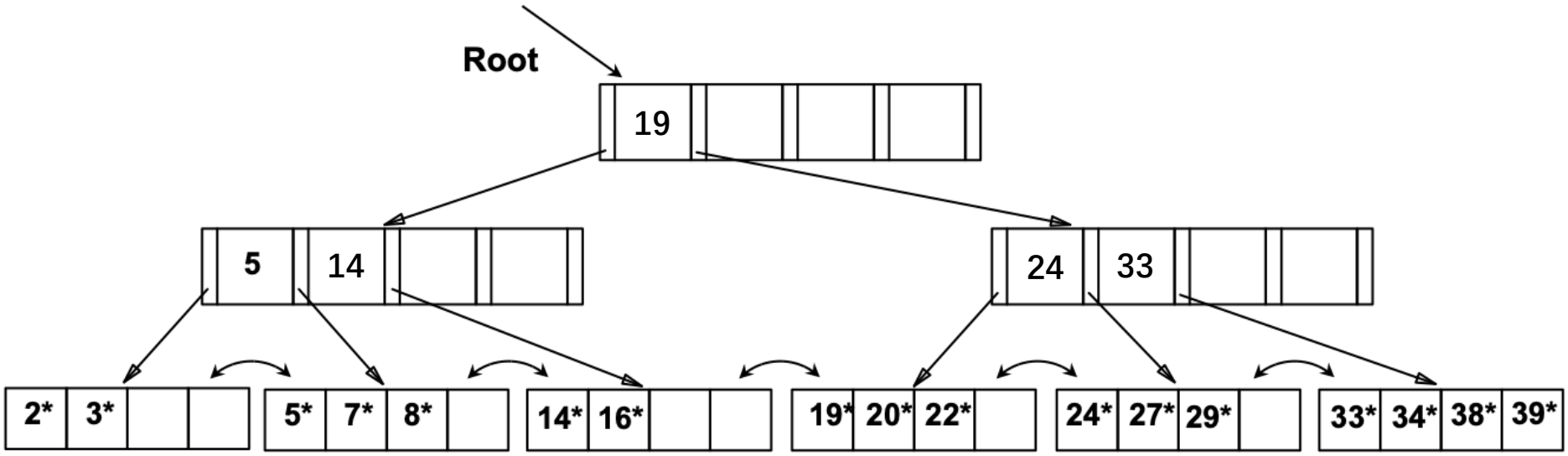


- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Review: B+ Tree Deletion

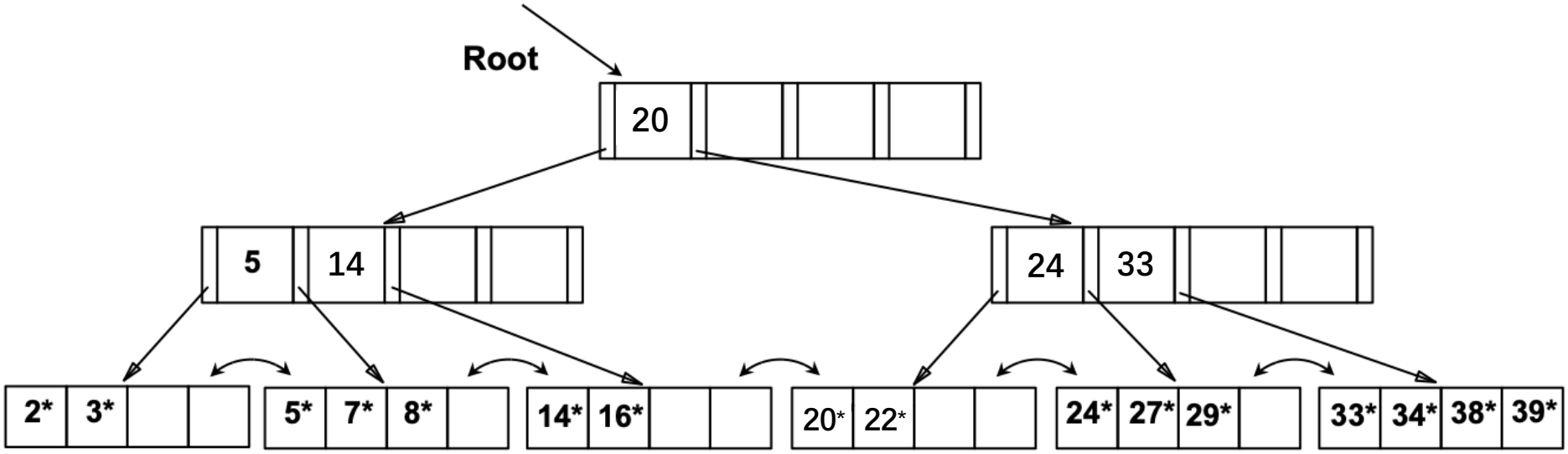
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Exercise1: Delete 19* and 20* From B+ Tree



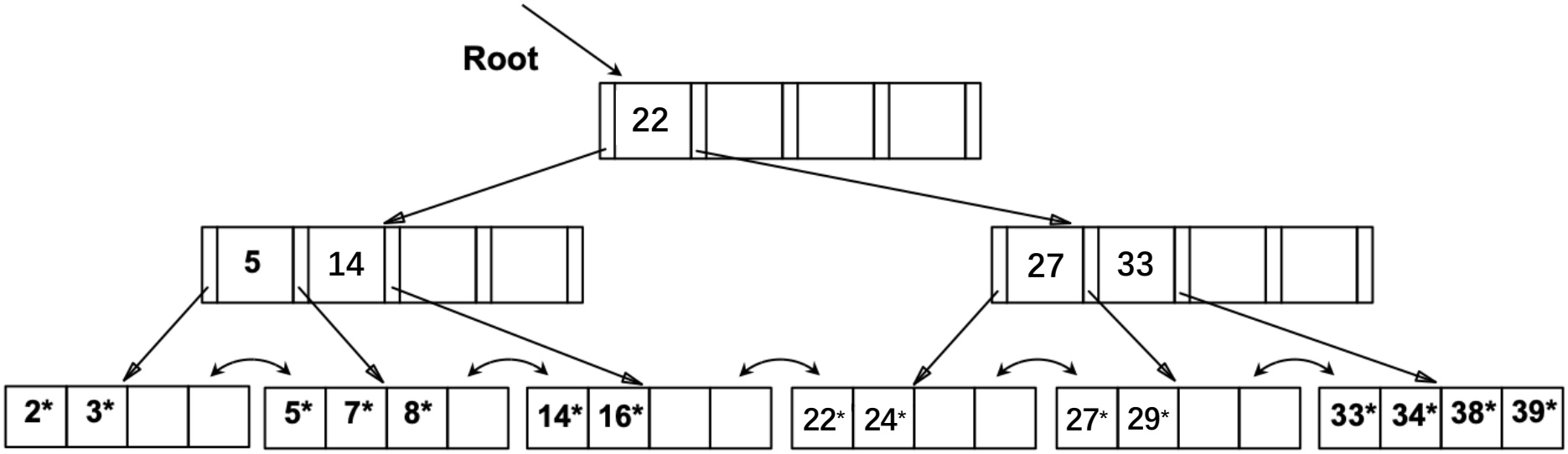
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

Delete 19



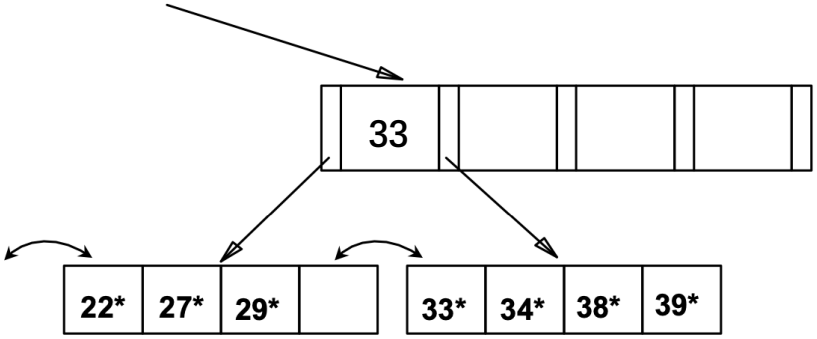
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

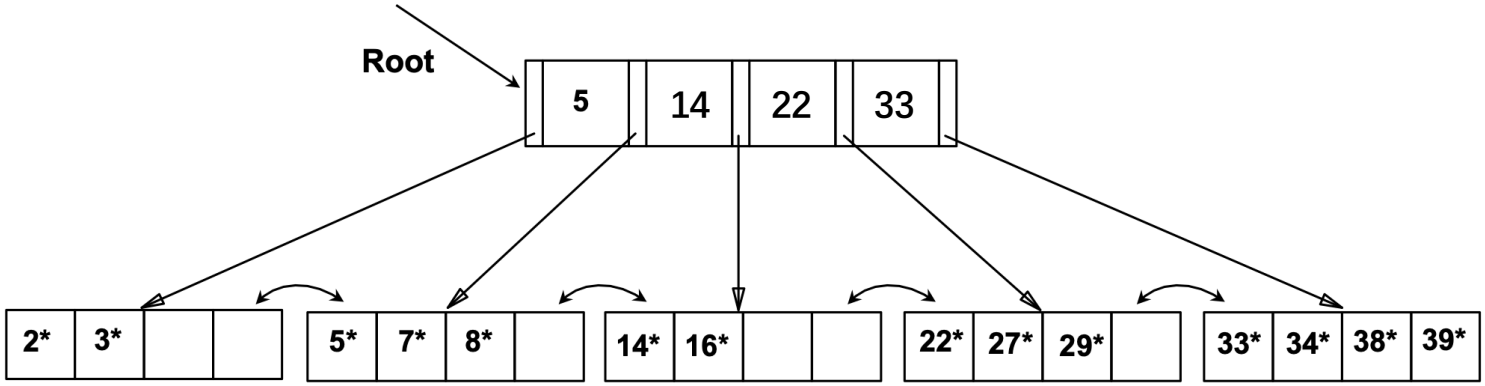
Delete 20



And Then Deleting 24* ...

- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).





Exercise 2.1: B+ Tree

Construct a B+ tree for the following set of key values:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Assuming that the tree is initially empty, values are added in ascending order, and the degree of both internal nodes and leaf nodes is 4. Thus each internal node in the tree can fit four pointers and three key values, denoted as $<A_1,K_1,A_2, K_2, A_3, K_3,A_4>$. Within each node, the keys satisfy $K_1 < K_2 < K_3$. For all search field values X in the sub-tree pointed by A_i, we have:

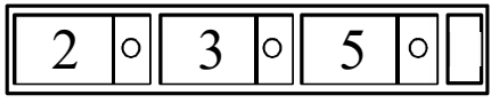
$K_{i-1} \leq X < K_i$ for $1 < i < 4$,

$X < K_i$ for $i=1$,

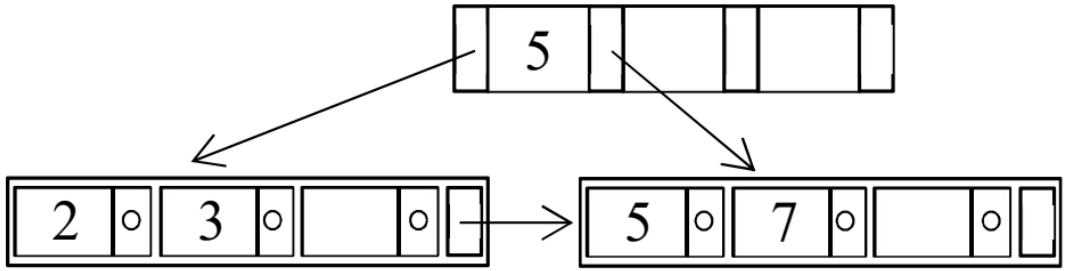
$K_{i-1} \leq X$ for $i = 4$.

Exercise 2.1: B+ Tree

- After inserting 2, 3, 5, the tree looks like

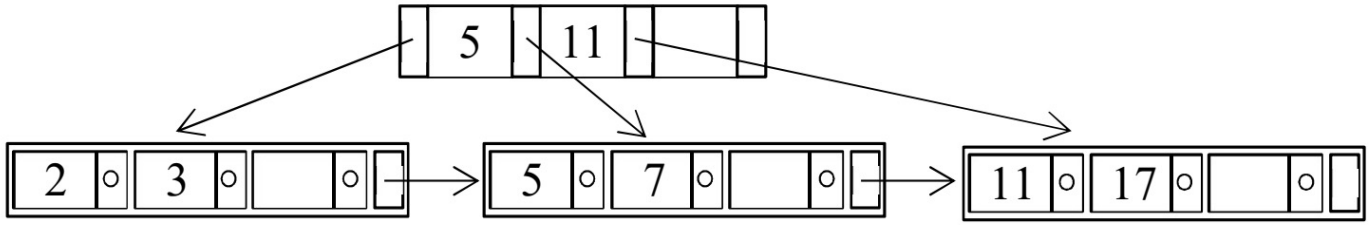


- After inserting 7, the tree looks like

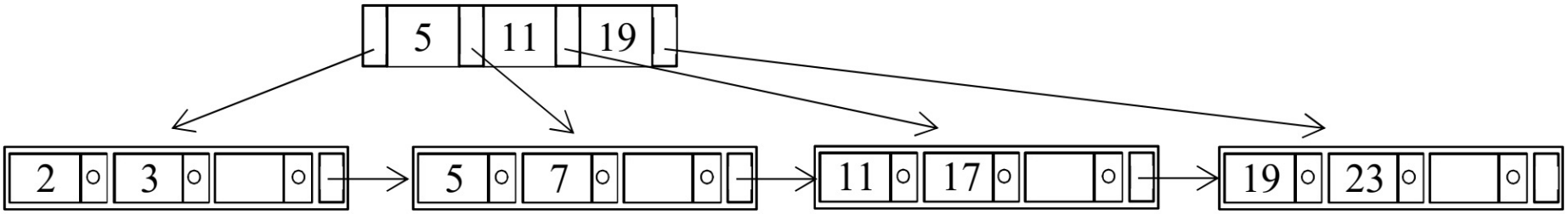


Exercise 2.1: B+ Tree

- After inserting 11, 17, the tree looks like

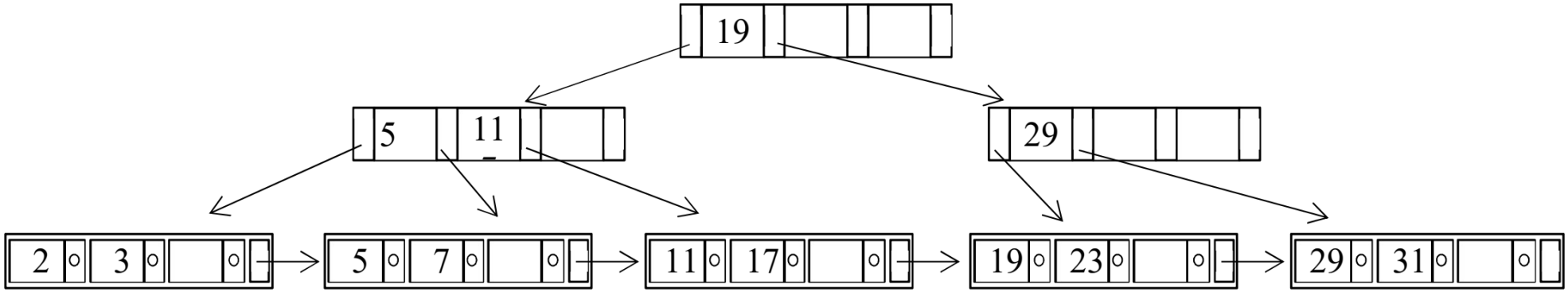


- After inserting 19, 23, the tree looks like



Exercise 2.1: B+ Tree

- After inserting 29, 31, the tree looks like



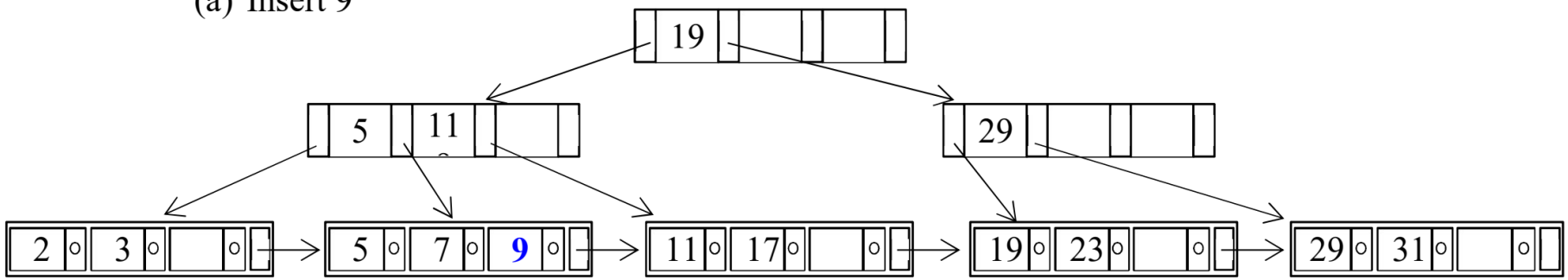
Exercise 2.2: B+ Tree

For the B+ tree constructed for question 1, show the form of the tree after each of the following series of operations:

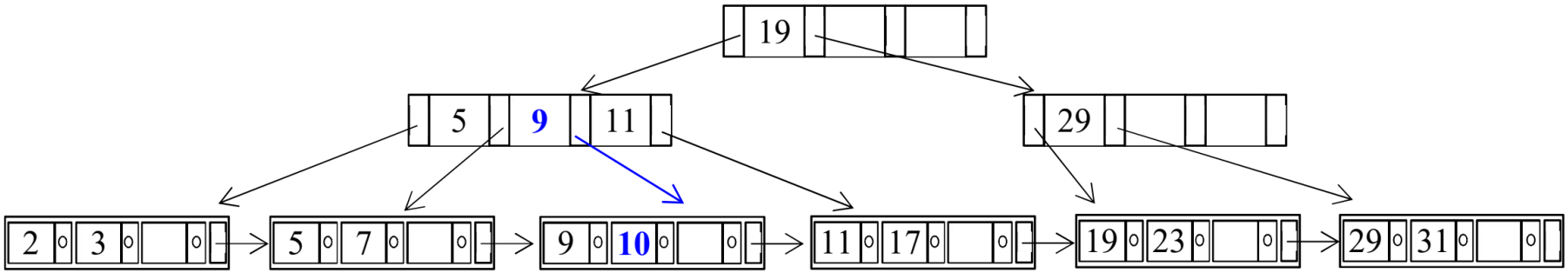
(a) Insert 9
(b) Insert 10
(c) Delete 19
(d) Delete 23
(e) Delete 31

Exercise 2.2: B+ Tree

(a) Insert 9

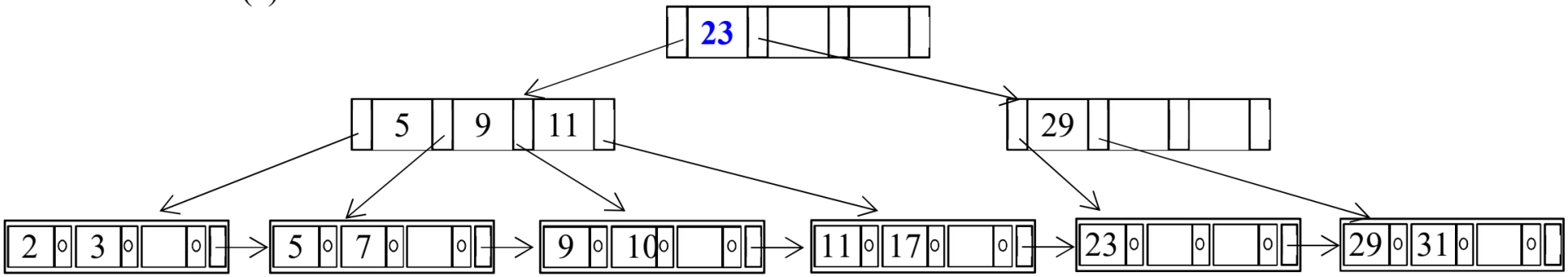


(b) Insert 10

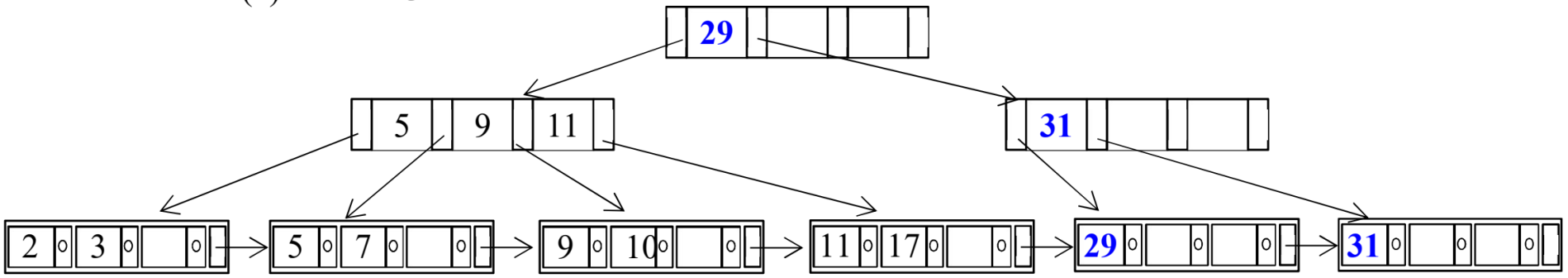


Exercise 2.2: B+ Tree

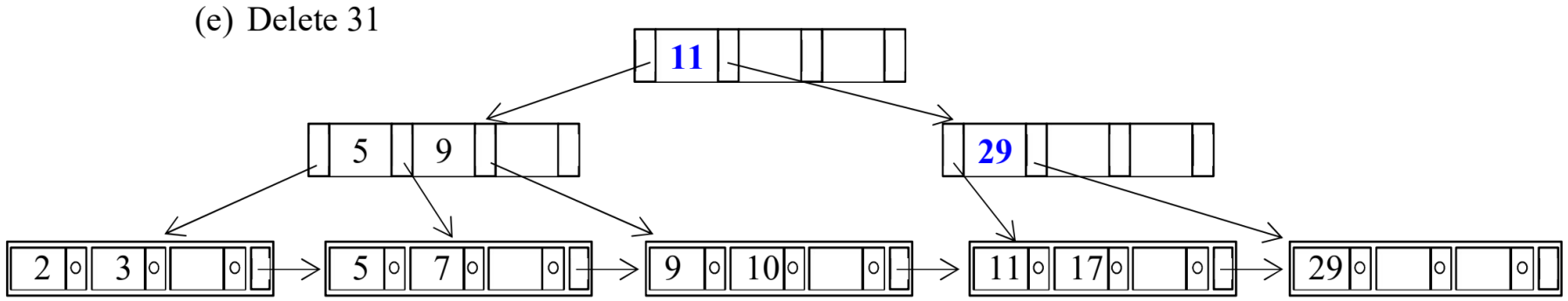
(c) Delete 19



(d) Delete 23



Exercise 2.2: B+ Tree



Regular Expressions

1. Repeaters (*, +, and { })

These symbols act as repeaters and tell the computer that the preceding character is to be used for more than just one time.

2. The asterisk symbol (*)

It tells the computer to match the preceding character (or set of characters) for 0 or more times (upto infinite).

```
Example : The regular expression ab*c will give ac, abc, abbc, abbbc….and so on
```

3. The Plus symbol (+)

It tells the computer to repeat the preceding character (or set of characters) at atleast one or more times(up to infinite).

```
Example : The regular expression ab+c will give abc, abbc,
abbbc, … and so on.
```

Regular Expressions

4. The curly braces { ... }

It tells the computer to repeat the preceding character (or set of characters) for as many times as the value inside this bracket.

```
Example : {2} means that the preceding character is to be repeated 2
times, {min,} means the preceding character is matches min or  more
times. {min,max} means that the preceding character is repeated at
least min & at most max times.
```

5. Wildcard (.)

The dot symbol can take the place of any other symbol, that is why it is called the wildcard character.

```
Example :
The Regular expression .* will tell the computer that any character
can be used any number of times.
```

Regular Expressions

6. Optional character (?)

This symbol tells the computer that the preceding character may or may not be present in the string to be matched.

```
Example :
We may write the format for document file as – “docx?”
The ‘?’ tells the computer that x may or may not be
present in the name of file format.
```

7. The caret (^) symbol (*Setting position for the match*)

The caret symbol tells the computer that the match must start at the beginning of the string or line.

```
Example : ^\d{3} will match with patterns like "901" in "901-333-".
```

8. The dollar ($) symbol

It tells the computer that the match must occur at the end of the string or before \n at the end of the line or string.

```
Example : -\d{3}$  will match with patterns like "-333" in "-901-333".
```

Regular Expressions

9. Character Classes

A character class matches any one of a set of characters. It is used to match the most basic element of a language like a letter, a digit, a space, a symbol, etc.

***\s**: matches any whitespace characters such as space and tab.*
***\S**: matches any non-whitespace characters.*
***\d**: matches any digit character.*
***\D:** matches any non-digit characters.*
***\w** : matches any word character (basically alpha-numeric)*
***\W**: matches any non-word character.*
***\b**: matches any word boundary (this would include spaces, dashes, commas, semi-colons, etc.*
***[set_of_characters]:** Matches any single character in set_of_characters. By default, the match is case-sensitive.*

```
Example : [abc] will match characters a,b and c in any string.
```

10. [^set_of_characters] *Negation:*

Matches any single character that is not in set_of_characters. By default, the match is case-sensitive.

```
Example : [^abc] will match any character except a,b,c .
```

Regular Expressions

11. [first-last] ***Character range:***

Matches any single character in the range from first to last.

```
Example : [a-zA-z] will match any character from a to z or A to Z.
```

12. The Escape Symbol (\ )

If you want to match for the actual '+', '.' etc characters, add a backslash(\ ) before that character. This will tell the computer to treat the following character as a search character and consider it for a matching pattern.

```
Example : \d+[\+-x\*]\d+ will match patterns like "2+2"
and "3*9" in "(2+2) * 3*9".
```

Regular Expressions

13. Grouping Characters ( )

A set of different symbols of a regular expression can be grouped together to act as a single unit and behave as a block, for this, you need to wrap the regular expression in the parenthesis( ).

```
Example : ([A–Z]\w+) contains two different elements of the regular
expression combined together. This expression will match any pattern
containing uppercase letter followed by any character.
```

14. Vertical Bar (|)

Matches any one element separated by the vertical bar (|) character.

```
Example :  th(e|is|at) will match words – the, this and that.
```

15. \number

Backreference: allows a previously matched sub-expression(expression captured or enclosed within circular brackets) to be identified subsequently in the same regular expression. \n means that group enclosed within the n-th bracket will be repeated at current position.

```
Example : ([a–z])\1 will match “ee” in Geek because the character
at second position is same as character at position 1 of the match.
```

Regular Expressions

16. Comment (?# comment)

Inline comment: The comment ends at the first closing parenthesis.

```
Example : \bA(?#This is an inline comment)\w+\b
```

17. # [to end of line]

X-mode comment. The comment starts at an unescaped # and continues to the end of the line.

```
Example :  (?x)\bA\w+\b#Matches words starting with A
```

Exercise: Regular Expression

- What the meaning of the following regular expression?
- ^([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$

Exercise: Dynamic Hashing

Bucket size: 3

Hash Function returns first X bits.

Key	binary	SampleData
01	0001	data1
07	0111	data2
03	0011	data3
08	1000	data4
12	1100	data5
14	1110	data6
11	1011	data7
02	0010	data8
10	1010	data9
13	1101	data10
04	0100	data11
09	1001	data12

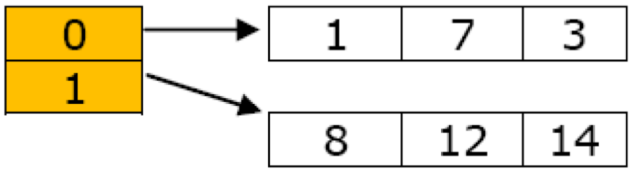
Each record consists of a **key** and **data**.

The binary representation of the key (using 4-bits) is provided for your convenience

Inserting Key Values – Dynamic Hashing

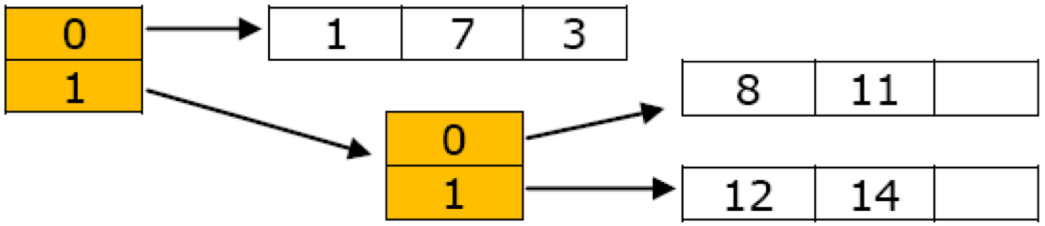
{ 1, 7, 3, 8, 12, 14, 11, 2, 10, 13, 4, 9 }

After inserting 1, 7, 3, 8, 12, 14

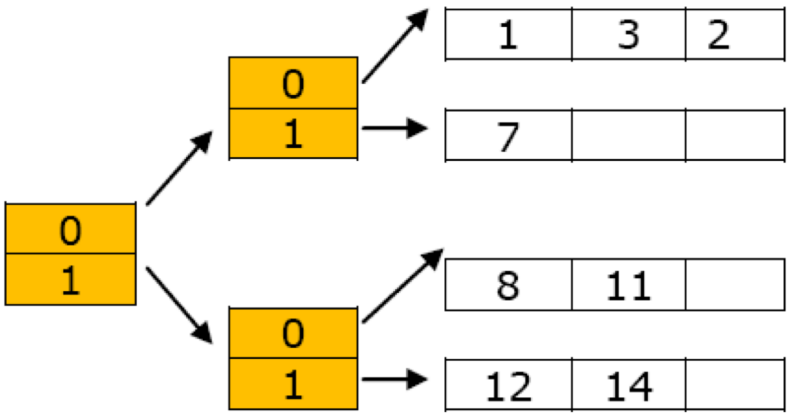



```
01 0001 data1
07 0111 data2
03 0011 data3
08 1000 data4
12 1100 data5
14 1110 data6
11 1011 data7
02 0010 data8
10 1010 data9
13 1101 data10
04 0100 data11
09 1001 data12
```

Inserting 11 produces overflow on bucket$_1$

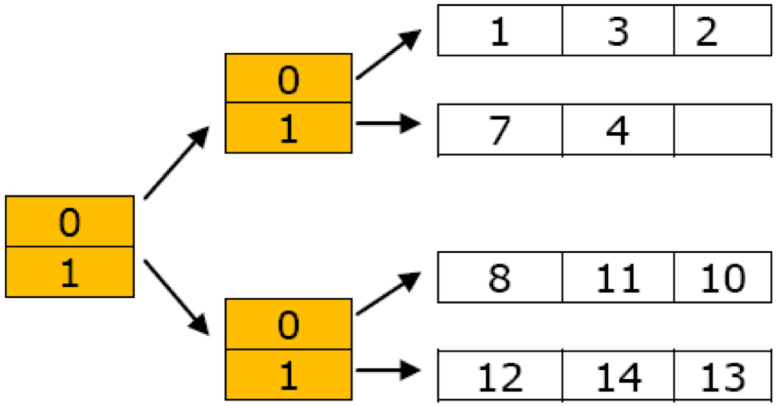


Inserting 2 produces overflow in bucket$_0$



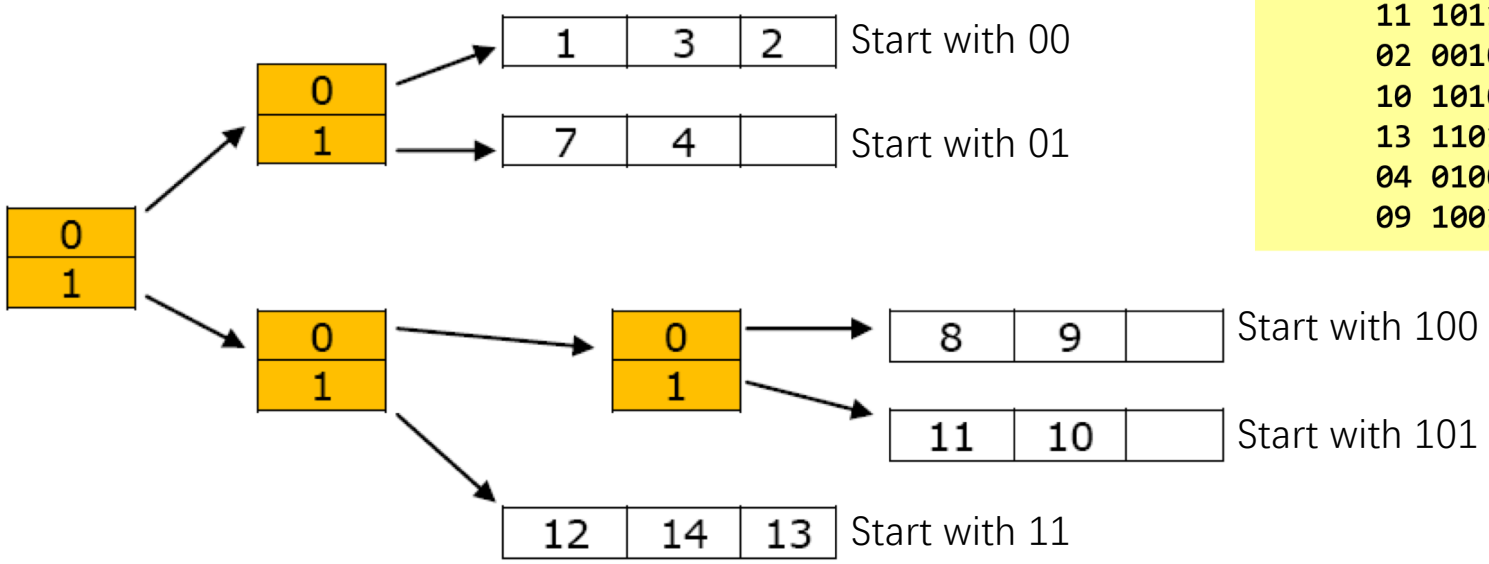

```
01 0001 data1
07 0111 data2
03 0011 data3
08 1000 data4
12 1100 data5
14 1110 data6
11 1011 data7
02 0010 data8
10 1010 data9
13 1101 data10
04 0100 data11
09 1001 data12
```

Inserting 10, 13, and 4



```
01 0001 data1
07 0111 data2
03 0011 data3
08 1000 data4
12 1100 data5
14 1110 data6
11 1011 data7
02 0010 data8
10 1010 data9
13 1101 data10
04 0100 data11
09 1001 data12
```

Inserting 9 produces overflow of Bucket$_{1,0}$



Thanks for your attention!