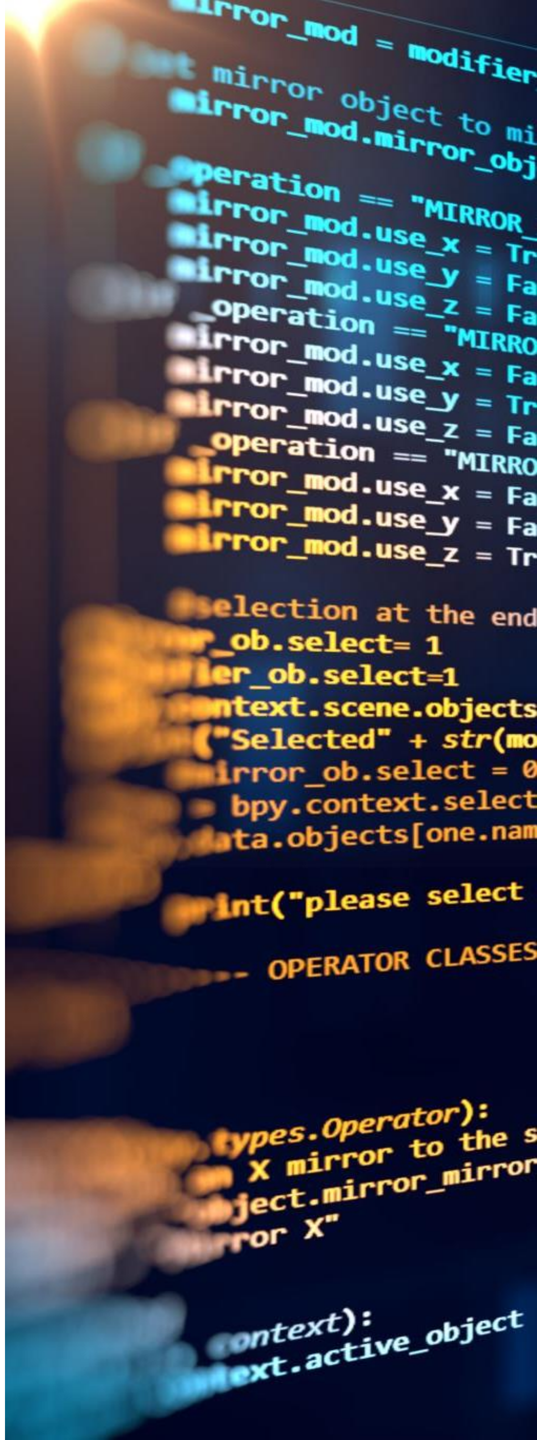
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

Multi-way Search Tree

Ahmed Ibrahim



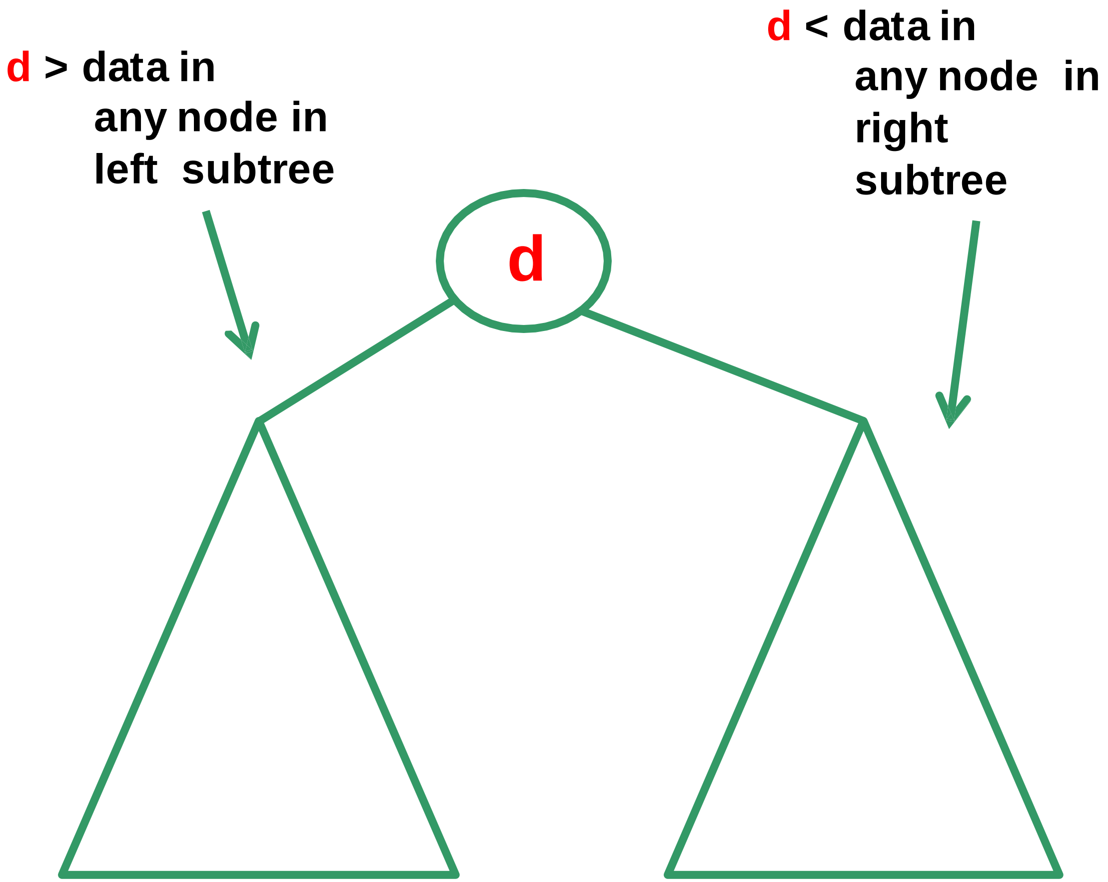
Recall: Binary Search Trees (BST)

- What is a Binary Search tree?

  A binary search tree is a **binary tree** in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

- Every BST is a BT, but every BT must not be a BST.

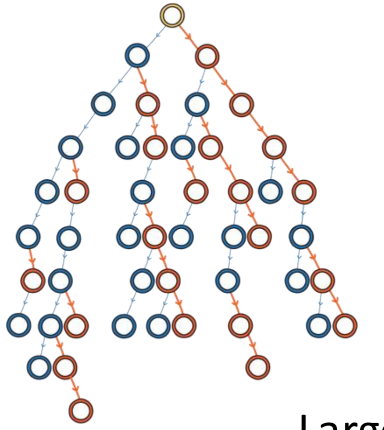
- There must be no duplicate nodes (in general).



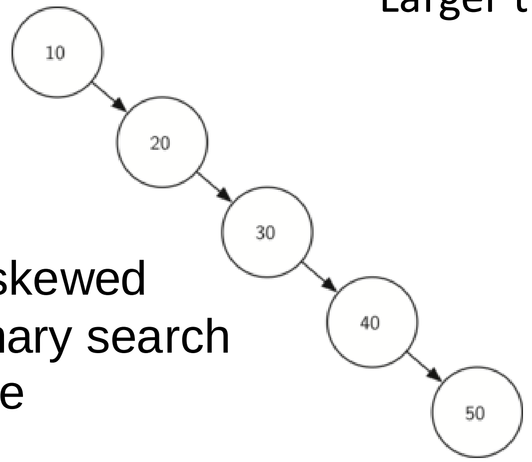
BTs Drawbacks

- In a BST, each node can have **only two children**, so the tree's height grows quickly as more nodes are added. For large datasets, this height increase makes searches slower, as it takes more steps to reach a **leaf** from the **root**.

- **Balance** is essential for good performance in a BST. If nodes are inserted in a sorted order (like ascending or descending), the tree can easily become unbalanced. Self-balancing trees (like AVL) solve this issue but add extra complexity and require rebalancing.



Larger tree



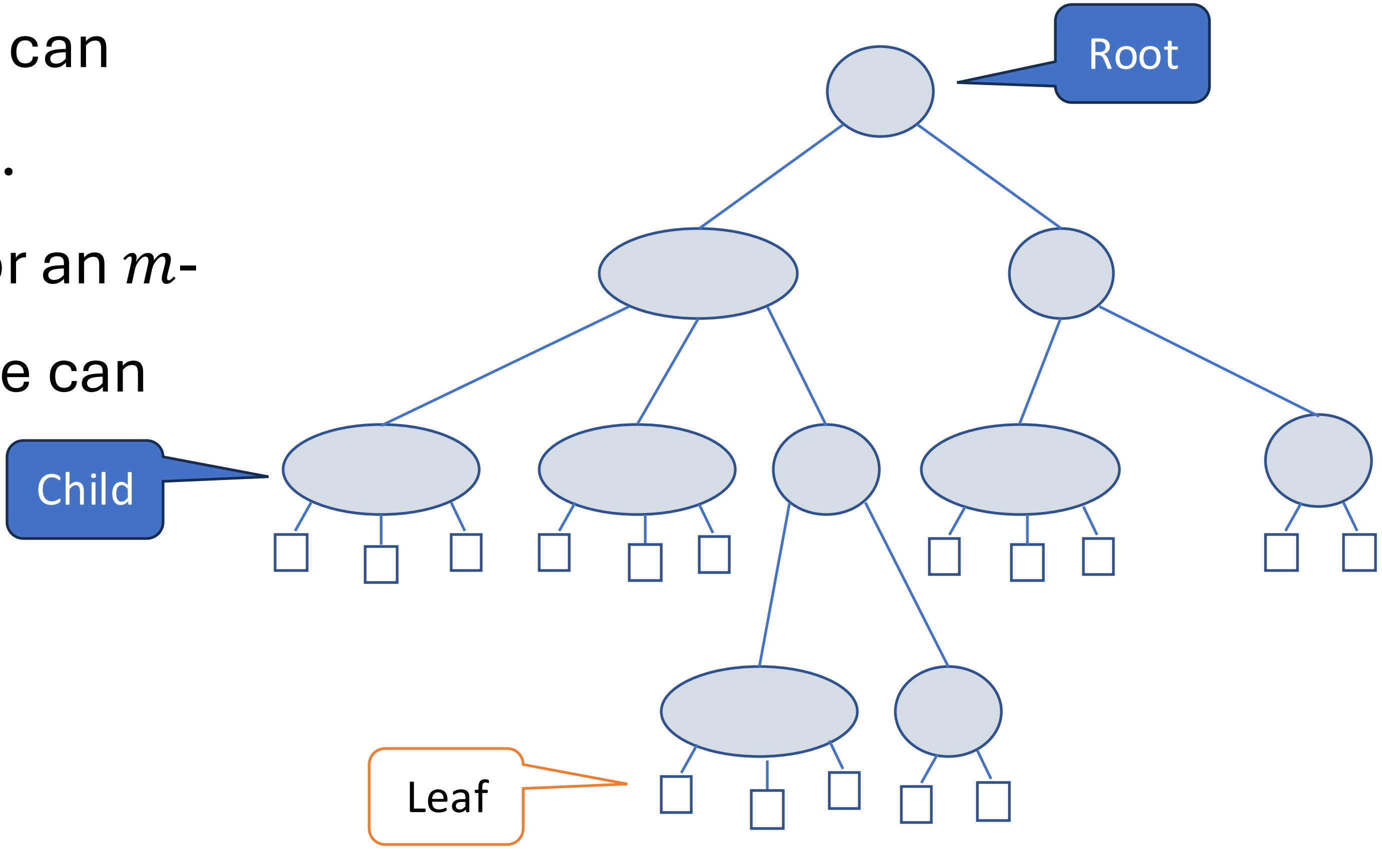
A skewed binary search tree

The concept of Multi-way Search Trees

- A multi-way tree is a tree that can have more than **two children**.
- A multi-way tree of order m (or an m-way tree) is one in which a tree can have m children.
- Legend:

Internal Node 

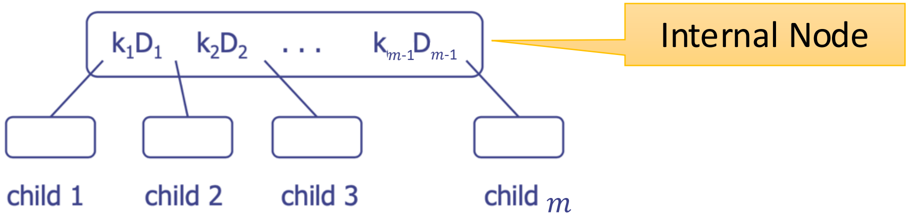
External Node 

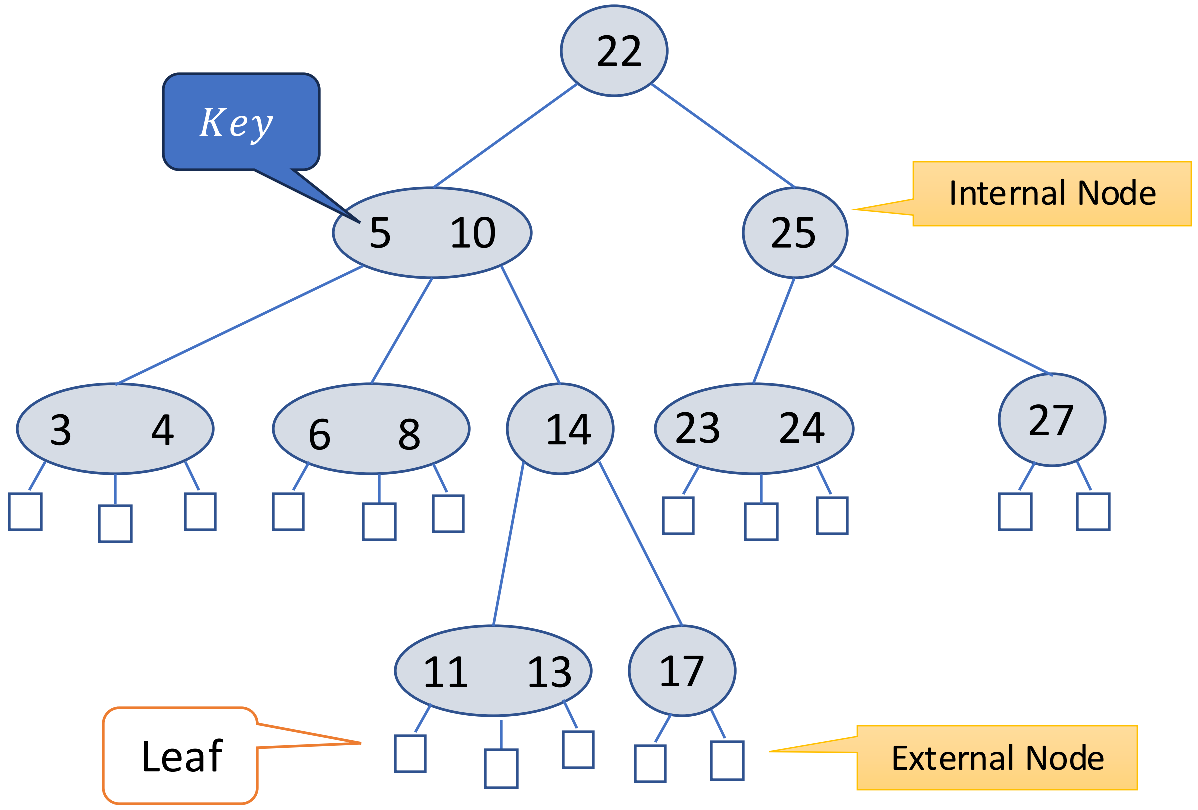


Example of 3-way tree

Multi-way Search Tree

- As with the other studied trees, the nodes in an m-way tree consist of $m-1$ ***entries*** and pointers to children.
- The **entries** are in the form of pairs (k, D) where k is the *key* and D is the value (*data*) associated with the *key*.
- The external nodes of a m-way search tree do not store any *entries* and are "*dummy*" nodes.





Example of 3-way tree

The Structure of Multi-Way Search Trees

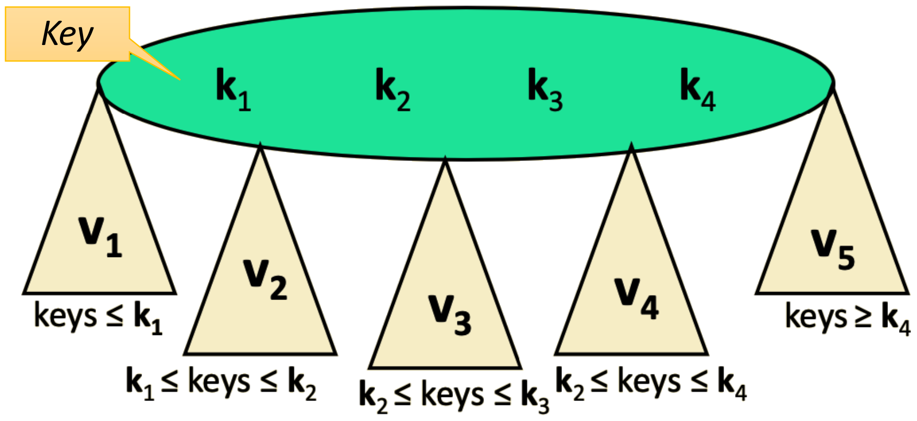
The following structure defines a simple m-way node structure, where m (>1) is a predefined constant.

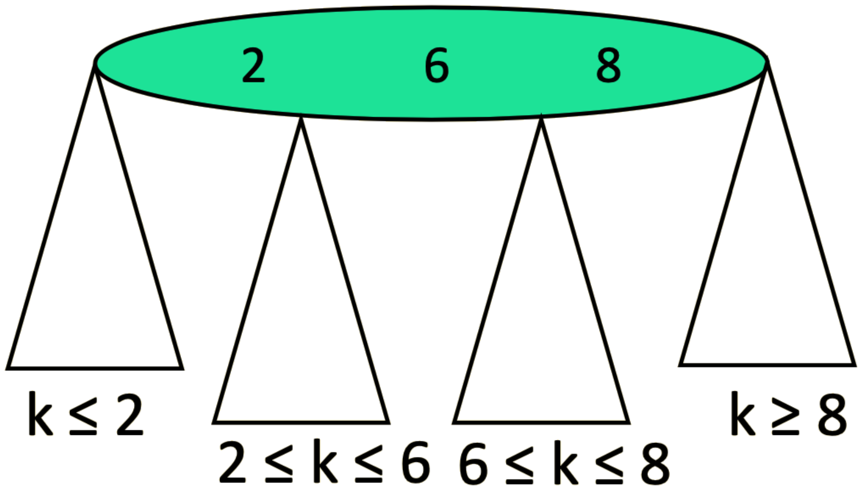
```
typedef struct node {
int count; // number of key values
int key[m-1]; // key arrays
struct node *child[m]; // sub-tree pointer array
} TNODE;
```

Properties of m-Way Search Trees

- A m-way search tree is an **ordered tree** such that
  - Each **internal node** has at least two and at most $\boldsymbol{m}$ children and stores $\boldsymbol{m\text{-}1}$ data items
  - **External nodes** have zero data items
  - Number of children = 1 + number of data items in a node
    - A node with three children is called a 3-node.

An Internal Node

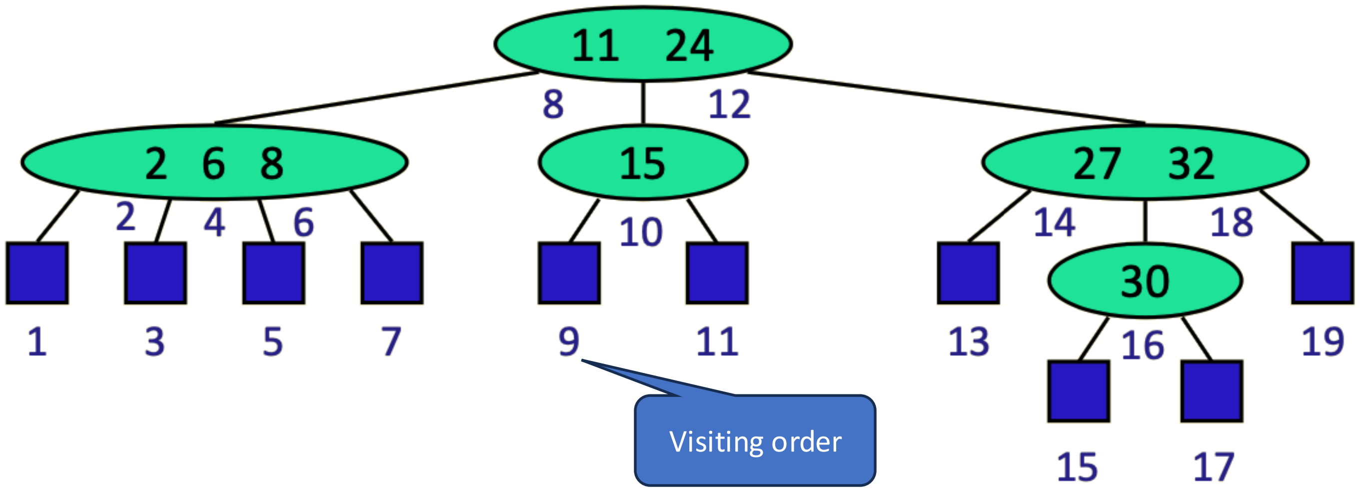


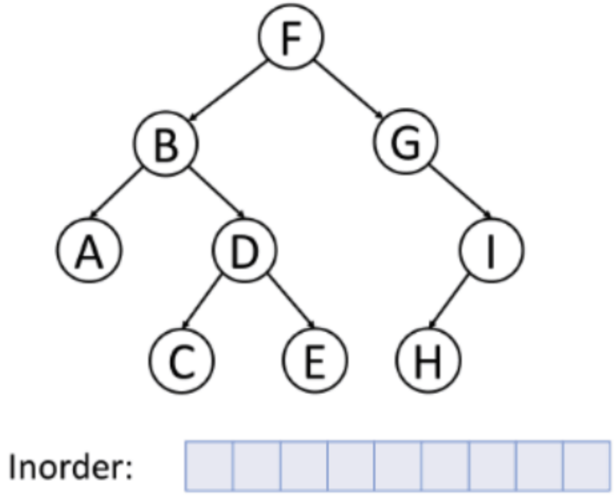


Numerical Example

- Each internal node has $m \geq 2$ children and stores m-1 entries.

Multi-Way Tree Traversal

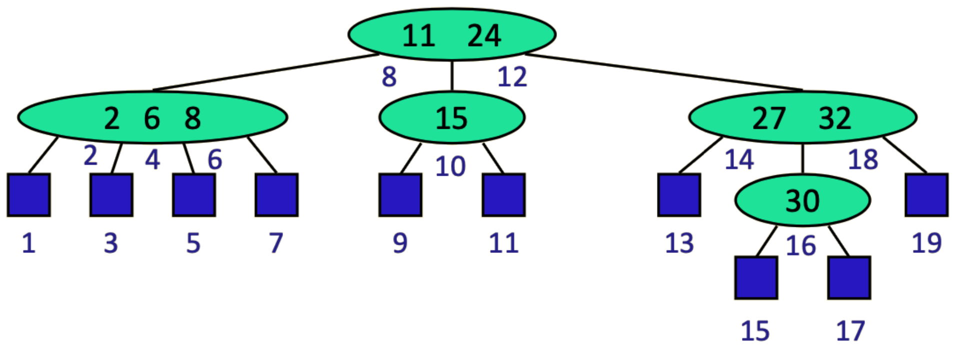




- In a multi-way search tree, in-order traversal involves visiting all keys in sorted order by **recursively** traversing child nodes and printing keys. Unlike BST, multi-way search trees may have more than two children per node, so the traversal is slightly different.
- An **in-order** traversal of a multi-way search tree visits the keys in **increasing order**.

Multi-Way Tree Traversal

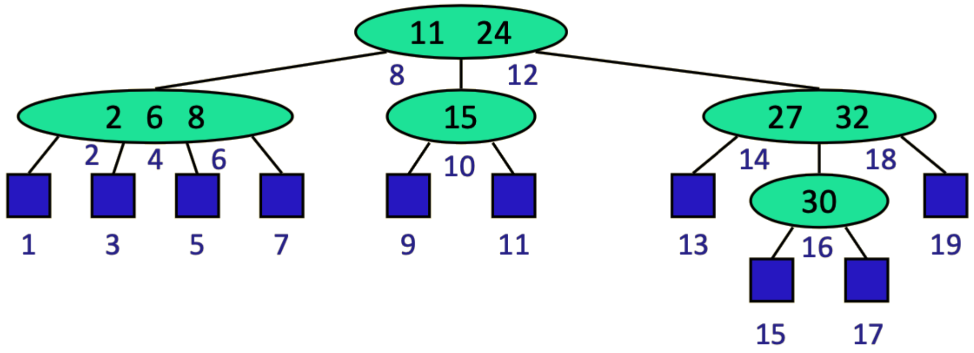
- Traversal in a BST visits each node about its two children (e.g., left subtree → node → right subtree in in-order).
- Traversal in a multi-way search tree involves visiting each key in the node and then recursively traversing each child subtree in the appropriate order. For example:
- In-order traversal in a multi-way tree involves:
  1. Recursively traversing the first child.
  2. Visiting the first key.
  3. Recursively traversing the second child.
  4. Visiting the second key.
  5. And so on for all keys and children.



Multi-Way Tree Traversal Algorithm

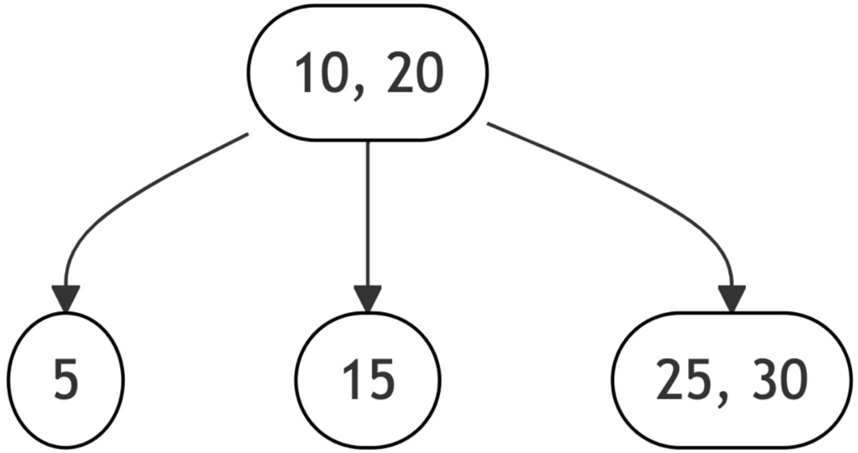
- The following program is an example of in-order traversal with printing key values.

```
/* in-order traversal of m-way tree*/
void print_inorder(TNODE *root) {
  if (root != NULL) {
  // Traverse the first child subtree
    print_inorder(child[0]);
    int i;
    for (i=0; i < root->count; i++) // Traverse through each key in the node
    {
      printf("%d ", root->key[i]; // Print the current key
      print_inorder(child[i+1]);} // Traverse the next child subtree
    }
}
```

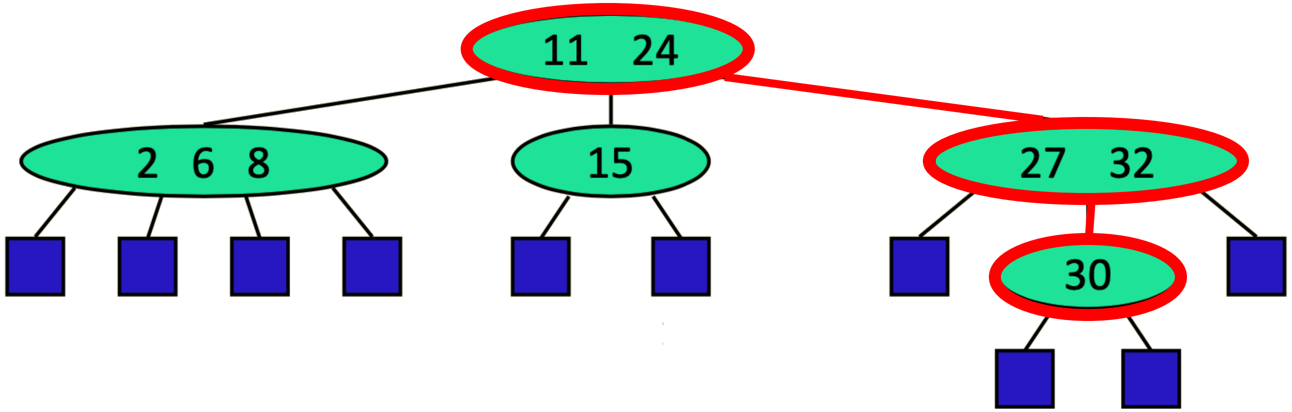


Example

- Consider a 3-way search tree with the following structure:
- **In-order traversal**:
- Traverse the leftmost child [5] → Visit key 10 → Traverse child [15] → Visit key 20 → Traverse child [25, 30].
- Result=> [5, 10, 15, 20, 25, 30].

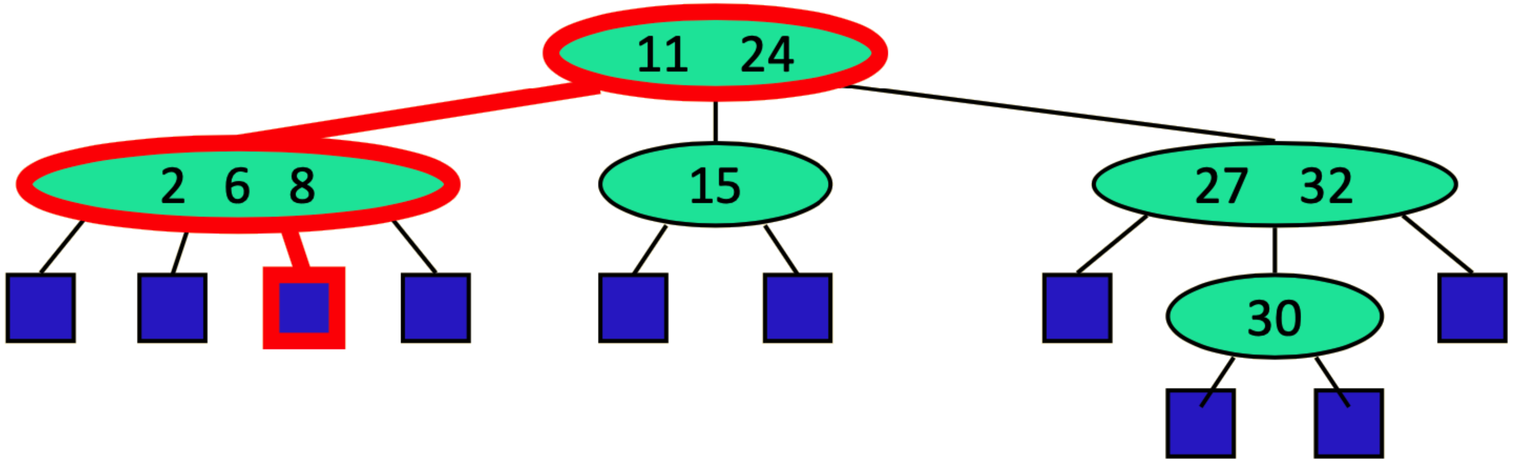


Multi-Way Search



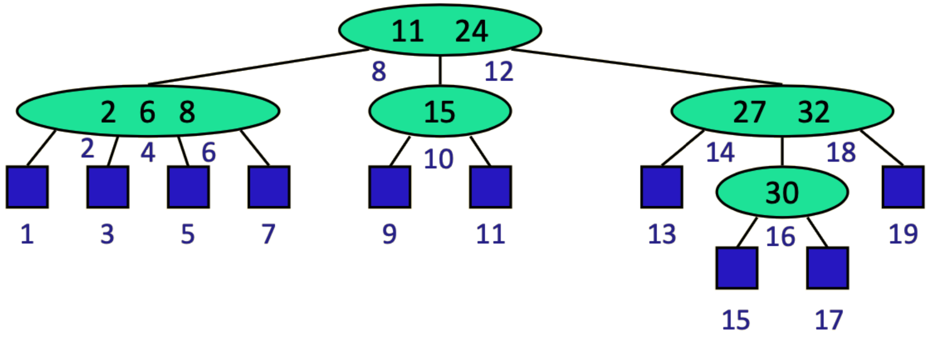
- Assuming $\boldsymbol{m}$ is a constant independent of a number of nodes, examining each node takes constant time (*O(1)*); thus, the time to search is **proportional** to the tree **height ($\boldsymbol{h}$)**.
- Within each node, searching among the keys takes O(1) time if the number of keys (m) is constant.
- Imagine that we are searching for k = 30

Another Example: Multi-Way Searching



- Search for key 7
  - Search terminates at a leaf child, which implies there is no entry with key 7

Multi-Way Searching Pseudocode



Algorithm get(r,k)

In: Root r of a multiway search tree, key k
Out: data for key k or null if k not in tree

if r is a leaf **then return** null

else {

Use binary search to find the index i such that either

- r.keys[i] = k, or
- r.keys[i] < k < r.keys[i+1]

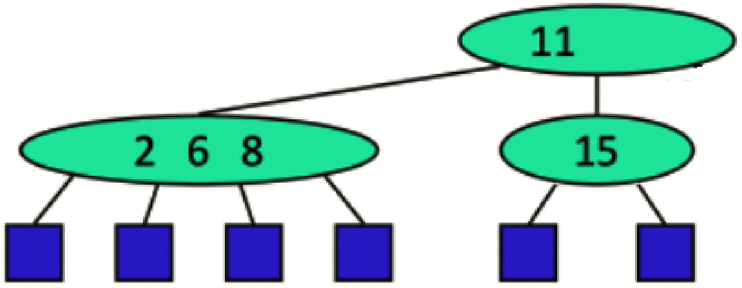
if k = r.keys[i] **then return** r.data[i]

else return get(r.child[i],k)

}

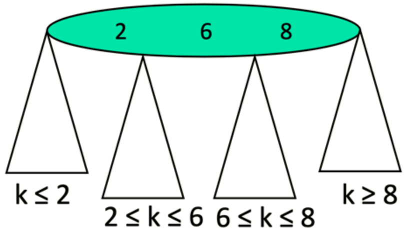
(2,4)-Tree

- (2,4) tree is a special m**-way search tree** with the following properties:
  - **Node-size property** – every internal node has at most **four children**. Every node can store between 1 and 3 keys.
  - **Depth property** – all **external nodes** have the **same depth** (the tree is balanced).
- Recall that in a multi-way search tree, the minimum number of children for a node is 2
  - Thus, a node can have 2, 3, or 4 children; thus a **(2,4) tree** is also called a **2-3-4 tree.**
- Used in databases and filesystems due to their efficient **search**, **insertion**, and **deletion**.

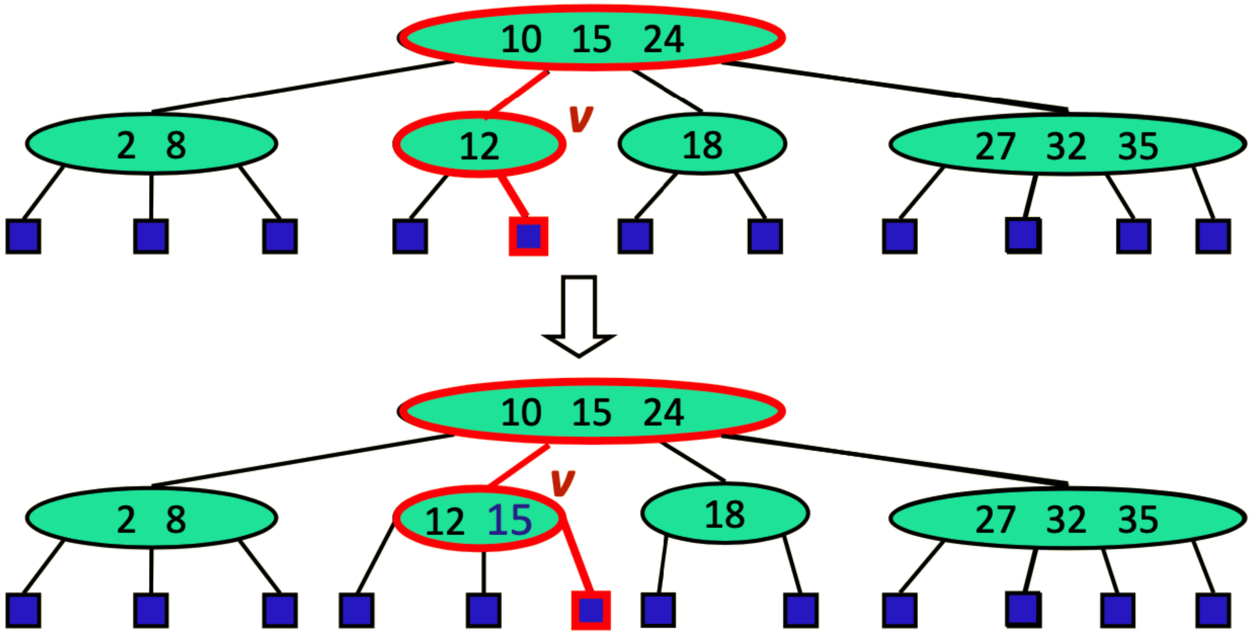


Insertion in (2,4) Trees

- How do we find the correct (preserving order) node v to insert?

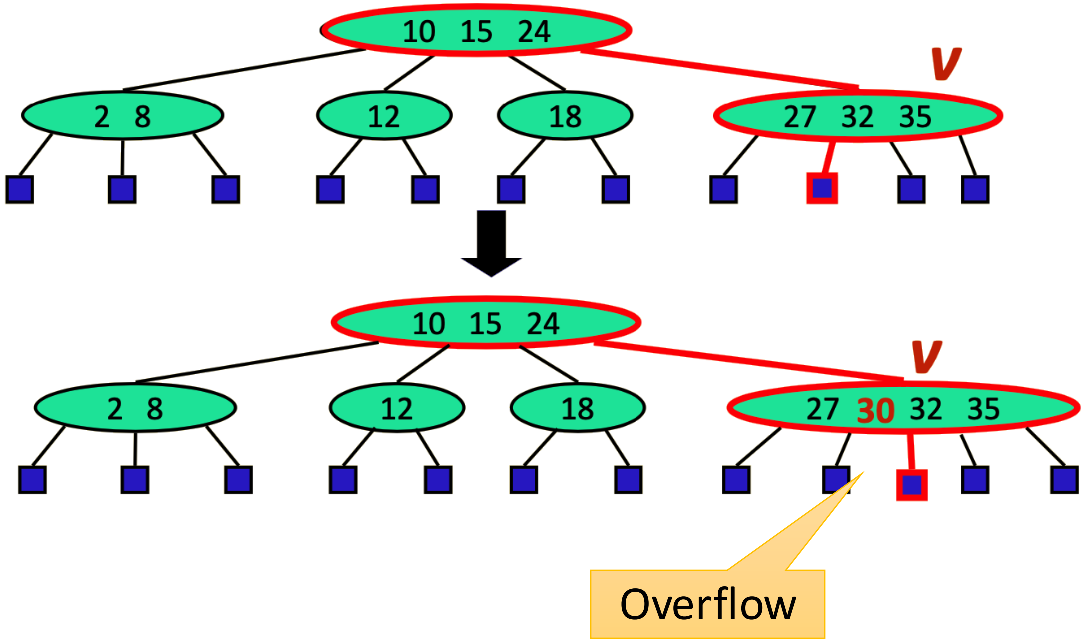


- ***Case 1*** **– key $\boldsymbol{k}$ is already in the tree**.
  1. Perform a search => O(*log* n)
  2. when reached node $\boldsymbol{v}$ storing $\boldsymbol{k}$, continue the search in the subtree to the of $\boldsymbol{v}$
  3. stop when reaching the node with only leaf children
- Example: insert 15

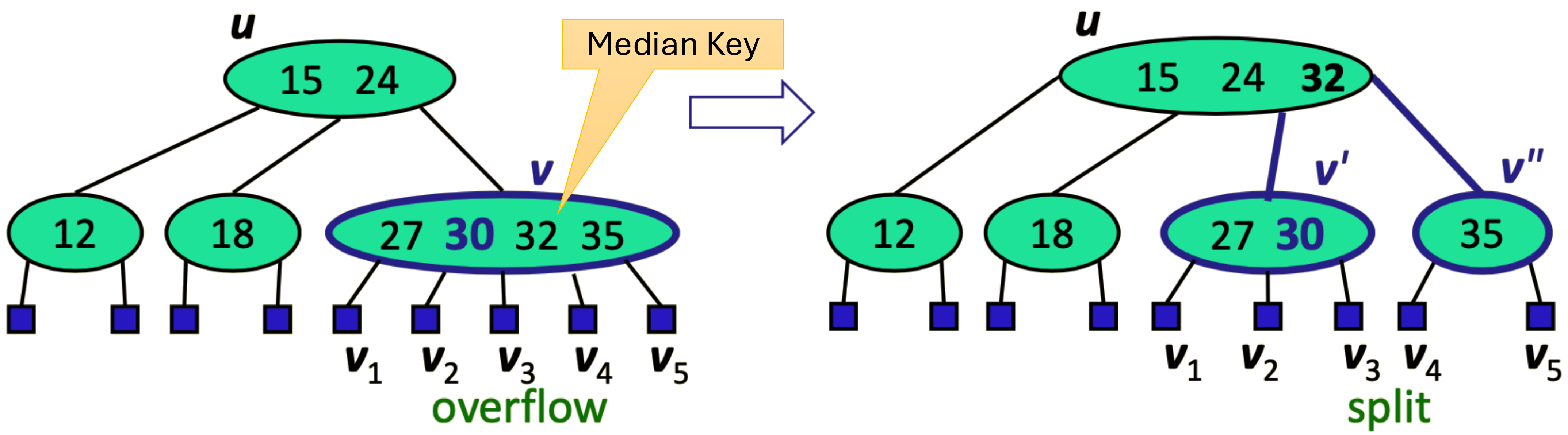


Insertion in (2,4) Trees

- How do we find the correct (preserving order) node $\boldsymbol{v}$ to insert?
- ***Case 2* – Key $\boldsymbol{k}$ is not in the tree.**
  1. Perform a search => O($log\, n$)
  2. If key k is not in the tree
  3. Then $\boldsymbol{v}$ is the leaf's parent reached by when searching for k.
- Example: insert 30
- However, there is a **violation** in the node size property
  - **Overflow** occurs when a 4-node becomes a 5-node, illegal in (2,4)-tree

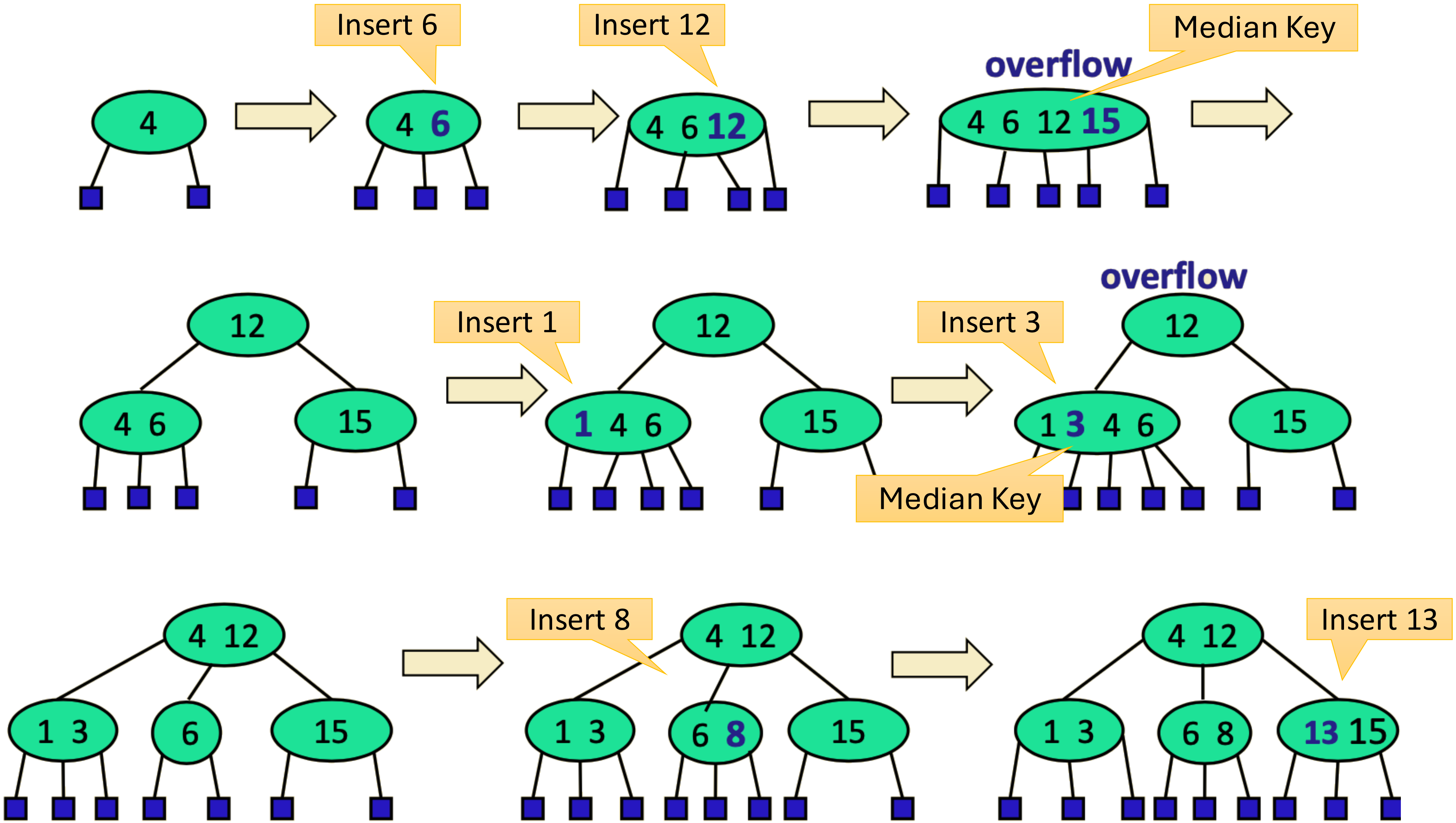


Insertion: Overflow and Split



- The median key is the *third* key in the sorted list. It splits the v node into two smaller nodes.
- *Overflow* may propagate to the parent node u.

Create (2,4) Trees



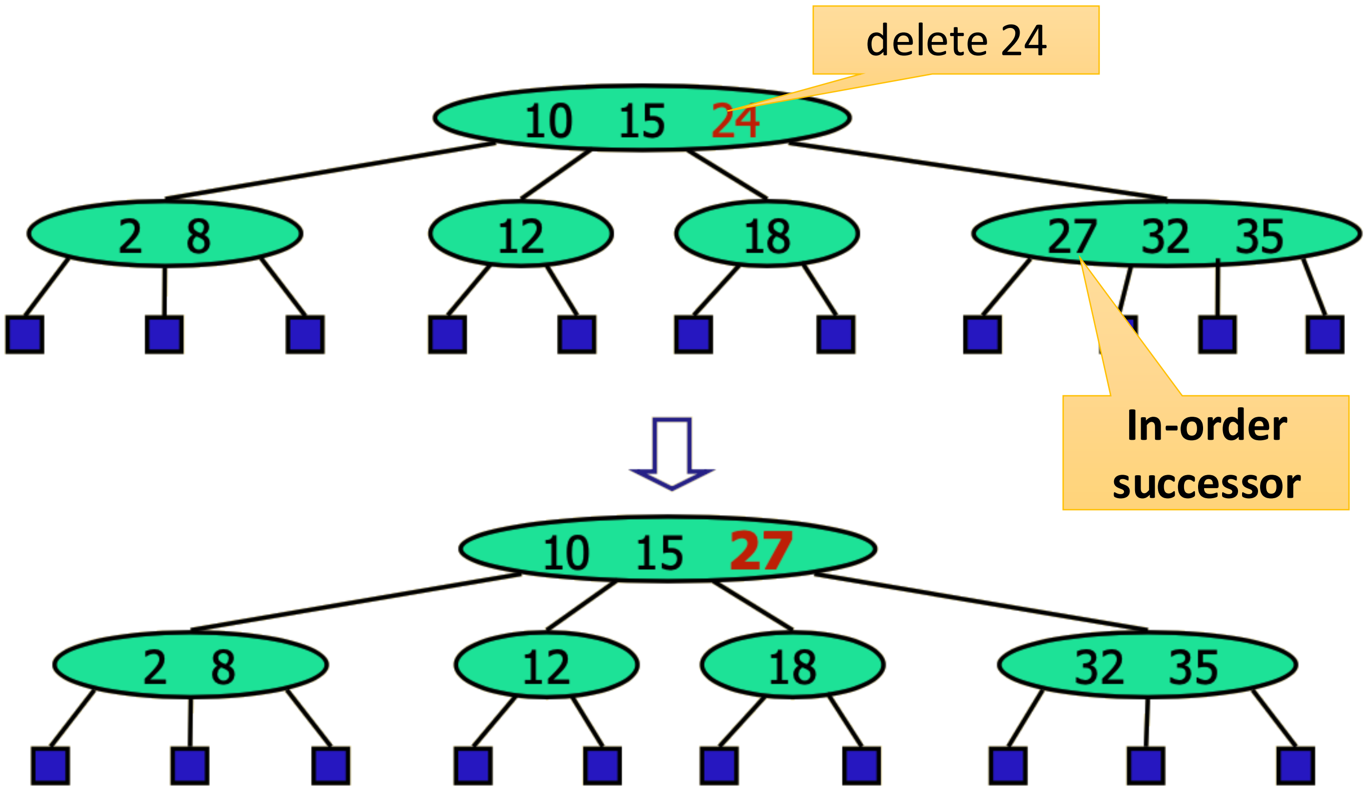
Insertion in (2,4) Tree Algorithm

1. **Start at the root of the tree (T)**.
   2. **If the root is full (contains 3 keys)**:
      a. Split the root into two nodes.
      b. Promote the *middle key* to create a new root.
      c. Update the tree so the new root has two children.
   3. **Find the correct leaf node where k belongs**:
      a. Traverse down the tree starting from the root.
      b. At each node, determine which child to move to based on the ranges defined by the keys.
      c. Repeat until a leaf node is reached.
   4. **Insert k into the leaf node**:
      a. Add k to the leaf node's appropriate position (keys remain sorted).

5. **Check for *overflow***:
   a. If the node now contains more than 3 keys:
      i. Split the node into two nodes.
      ii. Promote the *middle key* to the parent node.
      iii. Update the parent to reflect the new structure.
   6. **Repeat the overflow process**:
      a. If the parent node *overflows* after promotion, *split* the parent and promote its *middle key* to the next level-up.
      b. Continue this process recursively up the tree until no overflow occurs or a new root is created.

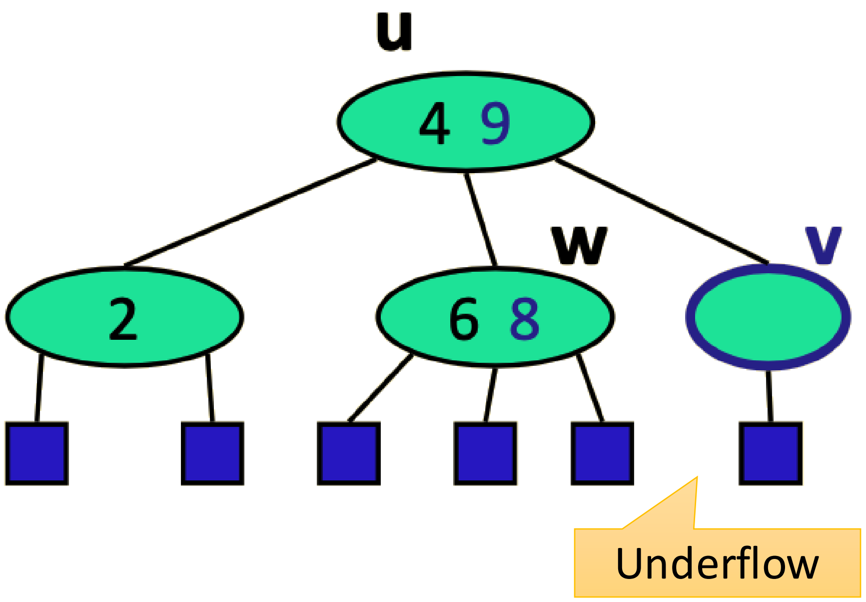
Deletion in (2,4) Trees

- *Case 1* – **If an entry is an internal node with no leaf children,** replace the entry with its **in-order successor** and delete the latter entry and one leaf.
- The **in-order successor** of a key in a (2,4)-tree is the smallest key larger than the given key, based on an in-order tree traversal.

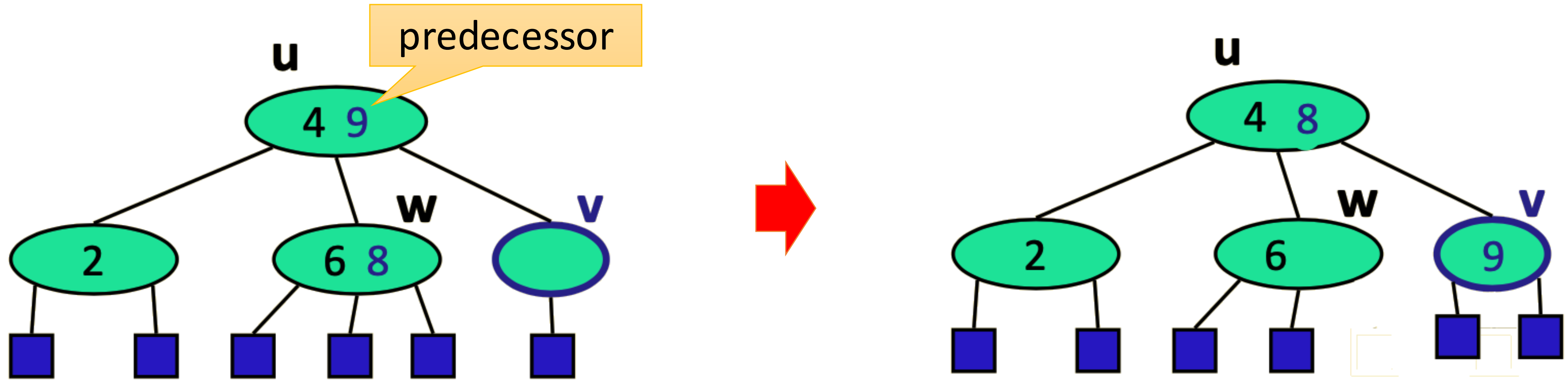


Underflow and Transfer

- *Case 2* – **Assume that the entry to be deleted is at node v with leaf children.**
- Deletion from v can cause an ***underflow*** (if v becomes a 1-node).
- To deal with ***underflow*** at node v with parent u, consider two cases:
  - **Transfer** operation, if an adjacent sibling w has at least **two entries** (*3 leaf nodes*)
  - **Fusion** operation, if all adjacent siblings of v are 2-node

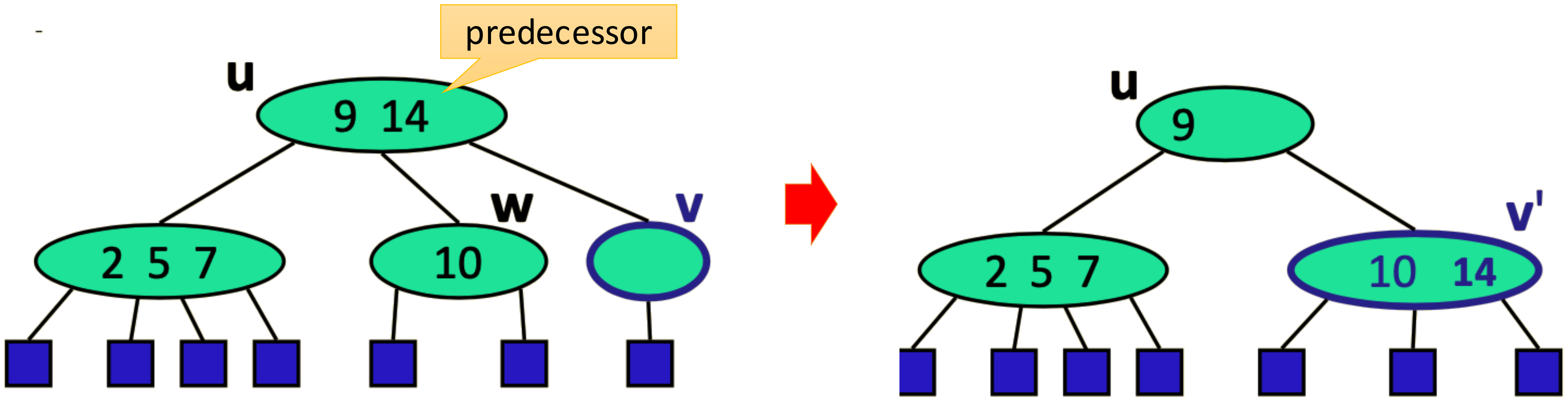


Transfer Operation



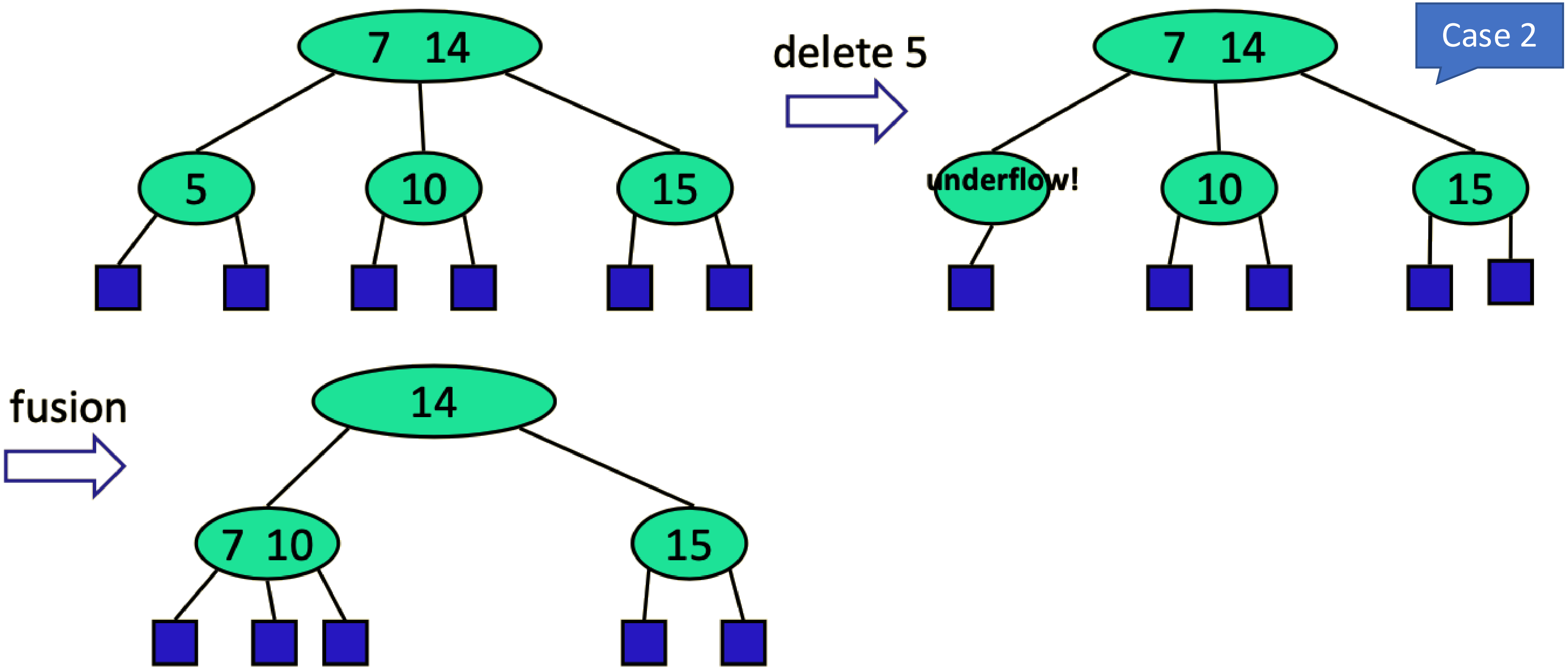
- An adjacent sibling w of v is a 3-node or a 4-node
  - Move an entry from $\boldsymbol{u}$ (the entry "between" $\boldsymbol{w}$ and $\boldsymbol{v}$) to $\boldsymbol{v}$
  - Move an entry from $\boldsymbol{w}$ (the entry with the key closest to the deleted key in $\boldsymbol{u}$) to replace the missing entry of $\boldsymbol{u}$

Fusion Operation



- All adjacent siblings of v are 2-node
- Here, we are going to do a **fusion.** After a fusion, ***underflow*** may propagate to the parent u.

Another Example



Deletion in (2,4) Tree Algorithm

1. **Start at the root of the tree (T).**
   2. **Search for key k in the tree**:
      a. Traverse down the tree, checking each node for k.
      b. If k is found in an **internal node**:
         i. Replace k with its **in-order predecessor** (largest key in the left subtree).
         ii. Recursively delete the predecessor key from the corresponding subtree.
      c. If k is found in a **leaf node**, proceed to step 3.
   3. **Delete key k from the leaf node**:
      a. Remove k from the node.
      b. If the node still has at least 1 key, stop.
      c. If the node becomes *underfull* (0 keys), fix the deficiency described in step 4.

Deletion in (2,4) Tree Algorithm

4. **Fix *underfull* nodes (fewer than 1 key):**

   a. **Borrow from a sibling (Transfer)**:

      i. Check if a sibling node (adjacent child of the same parent) has more than 1 key.

      ii. Borrow a key from the sibling and move the corresponding parent's key into the deficient node.

   b. **Merge with a sibling(Fusion):**

      i. If no sibling has extra keys, merge the deficient node with a sibling.

      ii. Move the parent's key that separates the two siblings into the merged node.

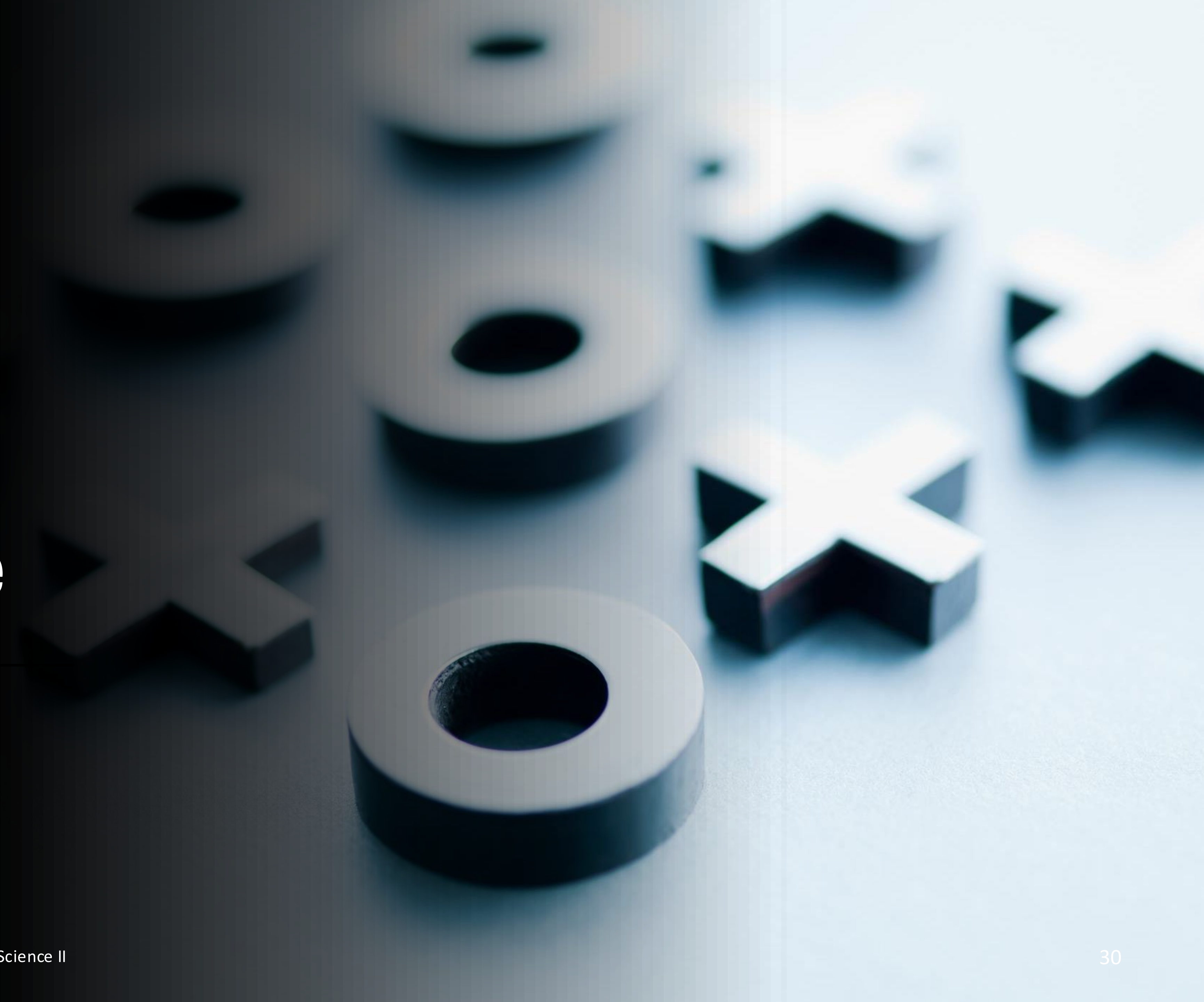
5. **Handle root *underflow***:

   a. If the root becomes *underfull* (0 keys) and has children:

      i. Promote the only child to become the new root.

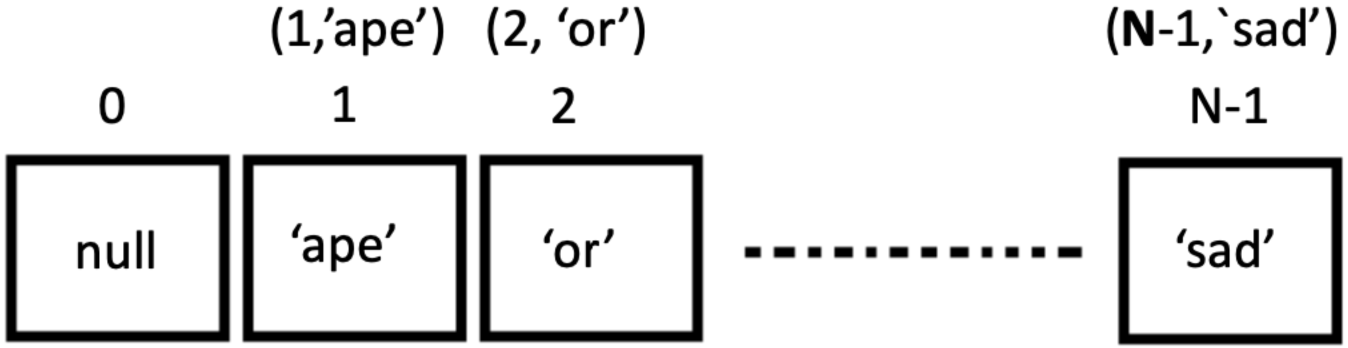
   b. If the root is empty and has no children, the tree becomes empty.

Hash Table



Key-Value Entries

- You have at most $\boldsymbol{N}$ entries $(\boldsymbol{k}, \boldsymbol{v})$
- Suppose keys k are unique integers between 0 to $N - 1$
- Create initially empty array A of size N
- Store (k, v) in A[k]
- Example



- Main operations (insert, find, remove) are O(1)
- We need O(N) space

Imagine that!

- What if we have 100 entries with integer keys, 0 to 1,000,000,000?
  - Do we still have O(1) insert(), delete(), find()?
  - We do not want 1,000,000,000 memory cells to store only 100 entries.
- What should we do?

The Concept of Hash Table

- Array of Fixed Size (***TableSize***)
- Each key is mapped into some number between 0 and (***TableSize - 1***). Mapping is done by something called the ***hash function***
- *The hash function ensures that two distinct keys are assigned to different cells.*
- Given the finite number of cells and an almost limitless supply of keys, a **hash function** is necessary to evenly distribute the keys among the cells!

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

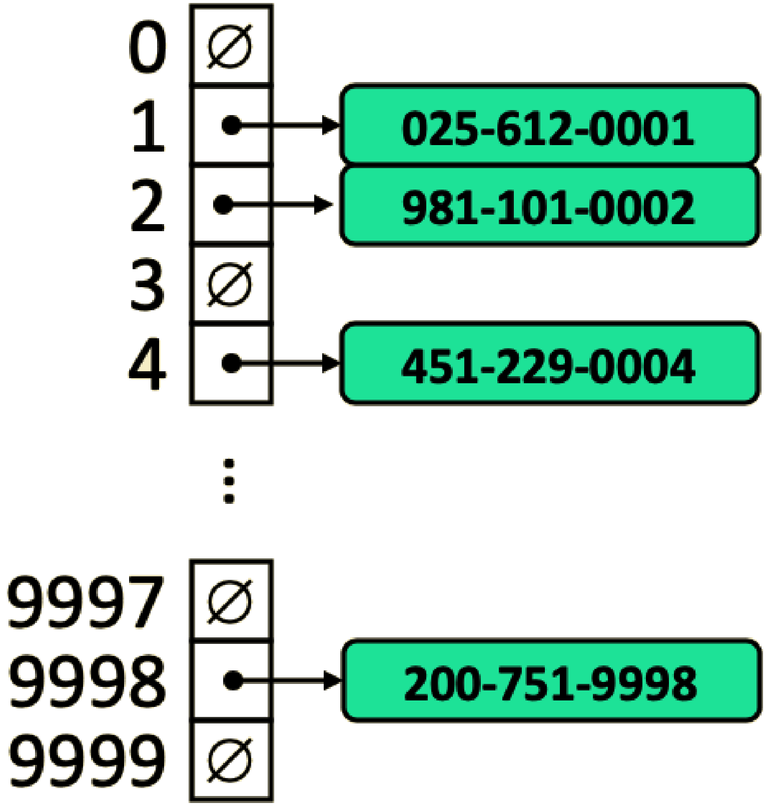
A Design Challenge

- Imagine a company has 5,000 employees and wants to store information about each employee using a hash table data structure.
- The company wants to use the employee's Social Security Number (SSN) of 9 digits as the key to looking up the corresponding employee information.
- The employee information includes the employee's **name**, **address**, **phone number**, and **salary**.
- The company needs to quickly look up an employee's information based on their **SSN**, as well as **add** and **remove** employees from the hash table.

Possible Solution

- Hash table for storing entries (SSN, info)
- The hash function h(x) => last four digits of SSN

- Thus, an array of size N => 10,000
  - The SSN is always of a fixed length.



Hash Tables and Hash Functions

- A hash table is a data structure that allows for efficient storage and retrieval of **key-value pairs** using a **hash function**.

- **Hash Tables**
  - **provide** fast data retrieval and insertion, typically in constant time, O(1), under ideal conditions.
  - **use** a hash function to transform input data (keys) into a fixed-size numerical value, determining where the data is stored in the table.

What if *keys* are NOT integers?

Key	Value
"Paul"	29
"Jane"	35
"Chloe"	88
"Alex"	18

Hashing Non-Integer Keys

- Array **A** of size **N** = 8
- Design function $h(k)$ that maps key $\boldsymbol{k}$ into integer range $0, 1, \ldots, \boldsymbol{N} - 1$
- Entry with key k **is** stored at index $h(\boldsymbol{k})$ in the array **A**

How far ‘p’ from ‘a’ => 15 $h(k)$ => 15 mod 8 = 7

Hashing value

Distance = ASCII code of k−ASCII code of ’a’.

- ASCII code of `'p'` = 112.
- ASCII code of `'a'` = 97.
- Distance from `'p'` to `'a'`: $112 - 97 = 15$.

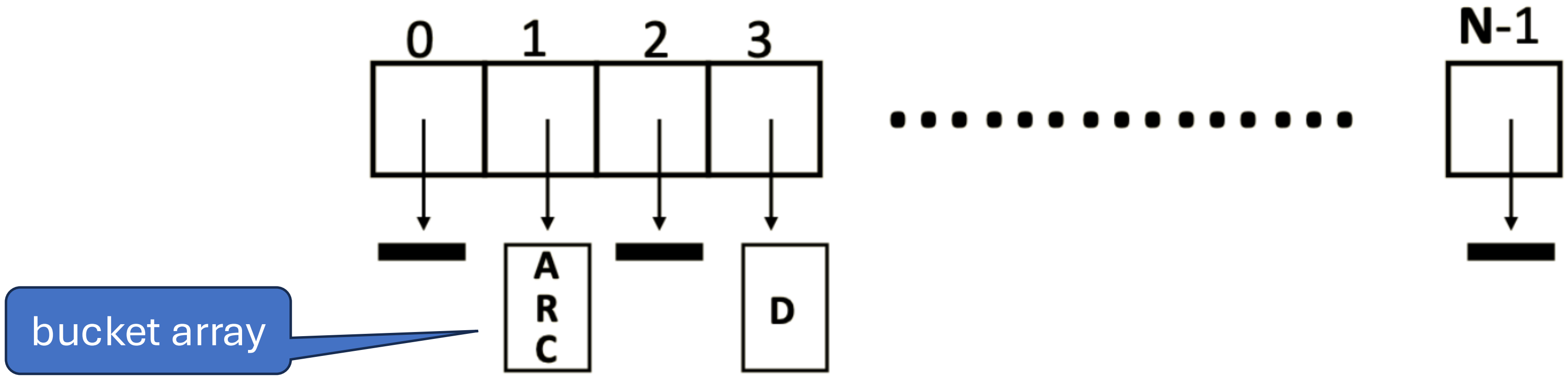
Key	Value
“Paul”	29
“Jane”	35
“Chloe”	88
“Alex”	18

Collision

- Collisions occur when different elements are mapped to the same cell.
- Collision resolution strategies
  - **Separate Chaining –** Store colliding keys in a **linked list** at the same hash table index
  - **Open Addressing –** Store colliding keys elsewhere on the table

Collision Resolution by Chaining

- What if you still have N keys, which may not be unique? (1, A) (1, R) (1, C) (3, D)

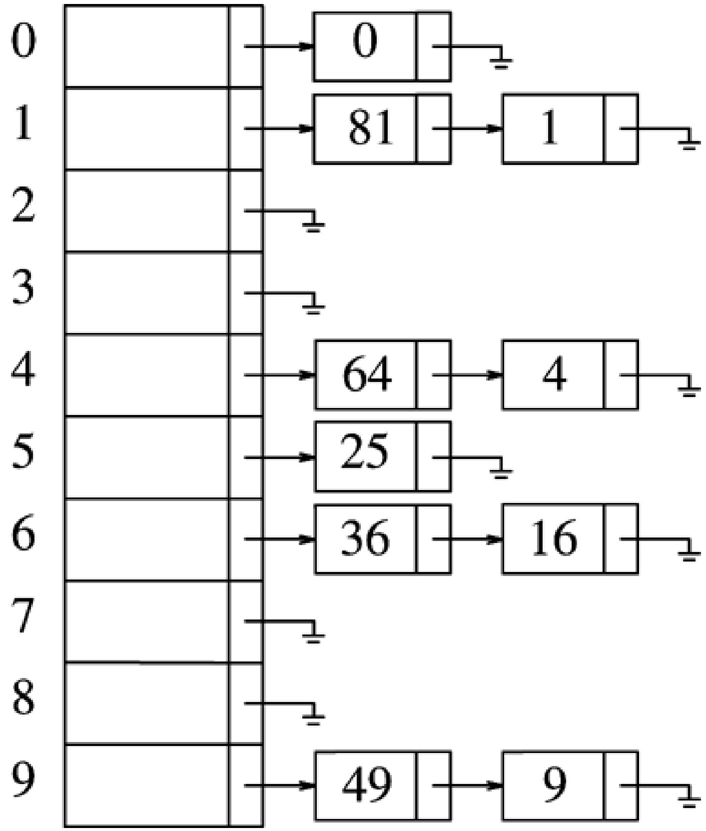


- A bucket array can be implemented as a **linked list.**
- Assume have at most a constant number of repeated keys methods find(), remove(), insert() are still O(1)

Example

- Hash table T is a vector of lists
  - Only singly linked lists are needed if memory is tight
- Key k is stored in the list at T[$h(k)$]
- E.g. TableSize = 10
  - $h(k)$ = k mod 10

Insertion sequence = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81



Load Factor λ

- The **load factor (λ)** is a measure that describes how **full a hash table** is.
- It is defined as: $\lambda = N / M$ Where:
    - N: The total number of elements stored in the hash table.
    - M: The total number of slots in the hash table.
- The average length of a chain is equal to the **load factor**
    - A smaller load factor indicates fewer collisions and better performance.
    - A larger load factor increases the likelihood of collisions, leading to longer chains.
- Ideally, we want $\lambda \leq 1$ (not a function of N)
- To maintain $\lambda \leq 1$, the hash table should resize (**rehash**) when it becomes **too full.**
- Keep the TableSize **prime** to ensure a good distribution

Example

- Imagine a hash table with M=10 (array slots) and N=7 elements.
- The load factor is: λ = N / M = 9 / 10 = 0.9
- We insert the following elements into the hash table= > Keys: 0, 81, 64, 25, 36, 49, 1, 4, 16
- With a hash function: $h(k)$ = k mod M (where M=10).
- **Collisions** may occur in buckets 0 and 6. We could consider resizing the table (rehashing) to reduce λ and minimize collision.

Index	Keys Stored (Chaining)
0	0, 81
1	1
2	
3	
4	4, 64
5	25
6	36, 16
7	
8	
9	49


Thank you