HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

Searching

CONTENT

- Sequential Search
- Binary Search
- Binary Search Tree
- Hashing

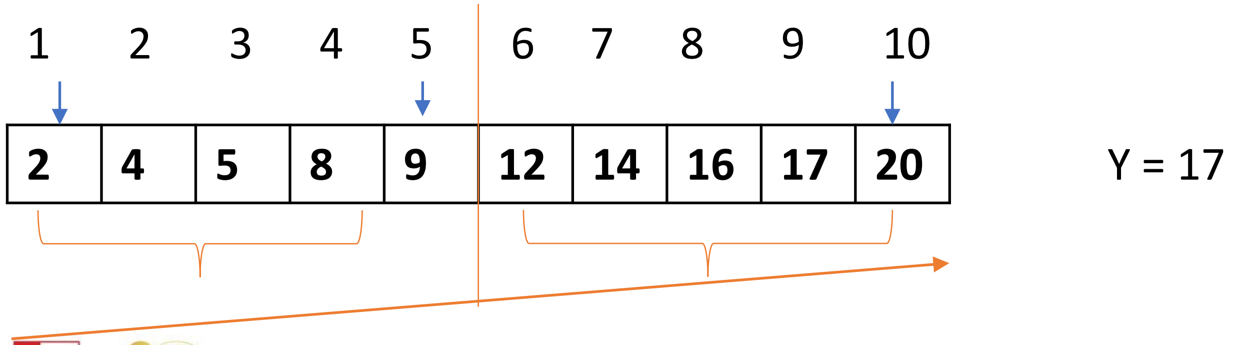
Sequential Search

- Given a sequence *X*[*L*..*R*] and a value *Y*. Find the index *i* such that *X*[*i*] = *Y*

```
sequentialSearch(X[], int L, int R,
    int Y) {
  for(i = L; i <= R; i++)
     if(X[i] = Y) return i;
  return -1;
}
```

Binary Search

- Sequence of objects is sorted in a non-increasing (non-decreasing) order of keys
- Base on divide and conquer:
    - Compare the input key with the key of the object in the middle of the sequence and decide to perform binary search on the left subsequence or the right subsequence of the object in the middle



Binary Search

- Sequence of objects is sorted in a non-increasing (non-decreasing) order of keys
- Base on divide and conquer:
    - Compare the input key with the key of the object in the middle of the sequence and decide to perform binary search on the left subsequence or the right subsequence of the object in the middle
- Running time: O(log(*R*-*L*))

```
binarySearch(X[], int L, int R,
     int Y) {
  if(L = R){
    if(X[L] = Y) return L;
    return -1;
  }
  int mid = (L+R)/2;
  if(X[mid] = Y) return mid;
  if(X[mid] < Y)
   return binarySearch(X,mid+1,R,Y);
  return binarySearch(X,L,mid-1,Y);
}
```

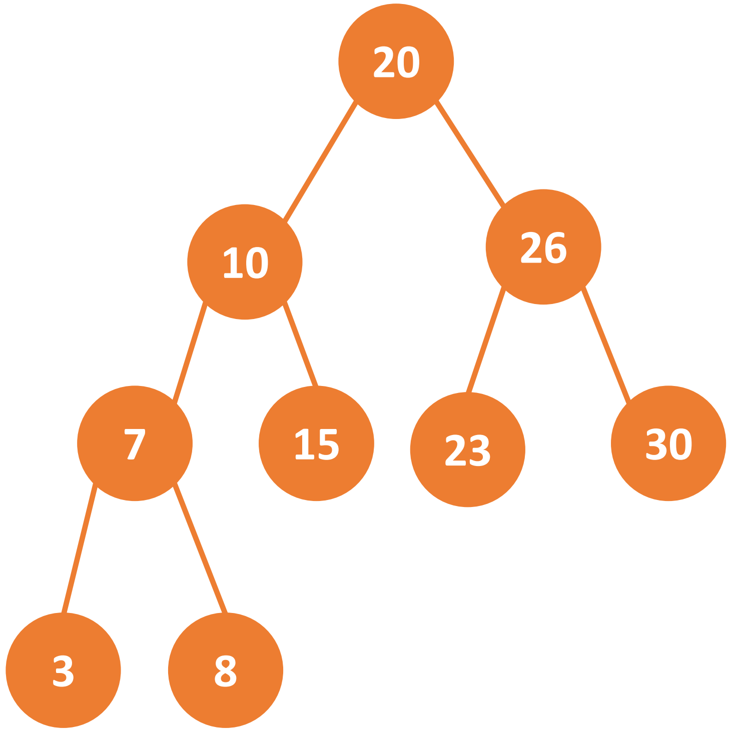
Binary Search

- **Exercise** Given a sequence of distinct elements a_1, a_2, …, a_N and a value b. Count the number of pairs (a_i, a_j) having $a_i + a_j = b$ ($i < j$)

Binary Search Tree - BST

- BST is a data structure storing objects under a binary tree:
  - Key of each node is greater than the keys of nodes of the left sub-tree and smaller than the keys of nodes of the right sub-tree

```
struct Node{
  int key;
  Node* leftChild;
  Node* rightChild;
};
Node* root;
```



Binary Search Tree - BST

- Operations
  - Node* makeNode(int v): create a node and return a pointer to the created node
  - Node* insert(Node* r, int v): create and insert a node with key v into the BST with root r
  - Node* search(Node* r, int v): find and return a node having key v in the BST with root r
  - Node* findMin(Node* r): find and return the node having minimum key
  - Node* del(Node* r, int v): remove the node having key v from the BST with root r

Binary Search Tree - BST

```
Node* makeNode(int v) {
  Node* p = new Node;
  p->key = v;
  p->leftChild = NULL;
  p->rightChild = NULL;
  return p;
}
```

```
Node* insert(Node* r, int v) {
  if(r == NULL)
    r = makeNode(v);
  else if(r->key > v)
    r->leftChild = insert(r->leftChild,v);
  else if(r->key <= v)
    r->rightChild = insert(r->rightChild,v);
  return r;
}
```

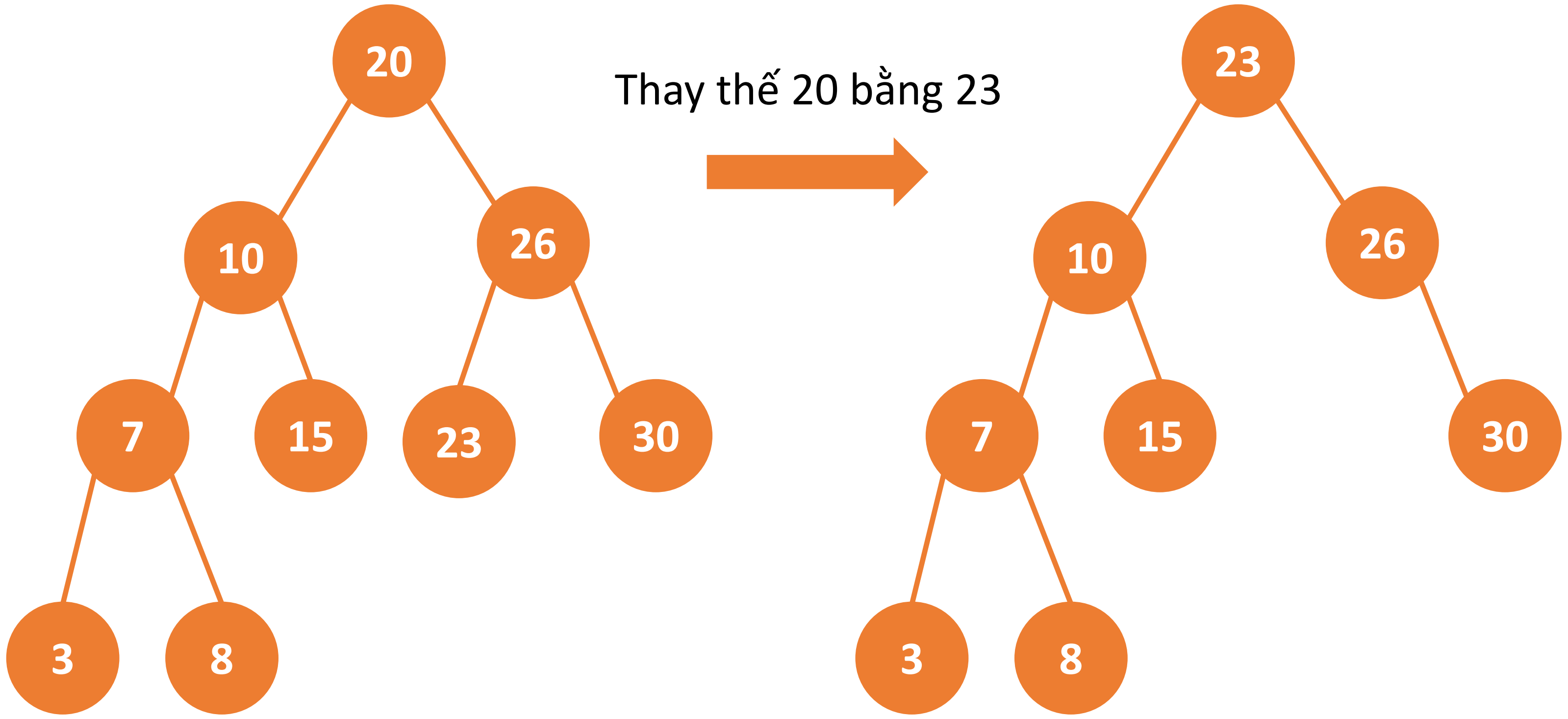
Binary Search Tree - BST

```
Node* search(Node* r, int v) {
  if(r == NULL)
    return NULL;
  if(r->key == v)
    return r;
  else if(r->key > v)
    return search(r->leftChild, v);
  return search(r->rightChild, v);
}
```

```
Node* findMin(Node* r) {
  if(r == NULL)
    return NULL;
  Node* lmin = findMin(r->leftChild);
  if(lmin != NULL) return lmin;
  return r;
}
```

Binary Search Tree - BST

- Xóa nút gốc



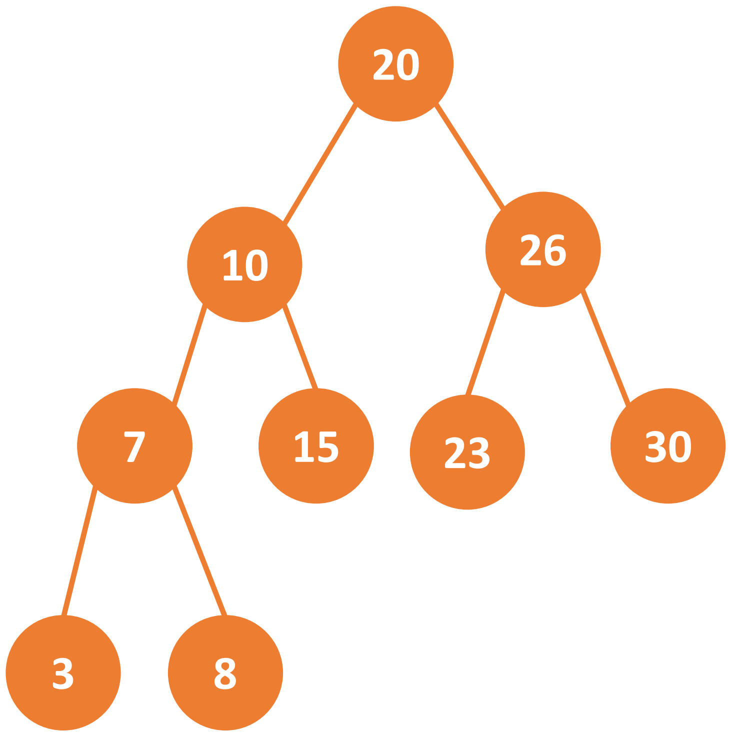
Binary Search Tree - BST

```
Node* del(Node* r, int v) {
  if(r == NULL) return NULL;
  else if(v < r->key) r->leftChild = del(r->leftChild, v);
  else if(v > r->key) r->rightChild = del(r->rightChild, v);
  else{
    if(r->leftChild != NULL && r->rightChild != NULL){
      Node* tmp = findMin(r->rightChild);
      r->key = tmp->key;  r->rightChild = del(r->rightChild, tmp->key);
    }else{
      Node* tmp = r;
      if(r->leftChild == NULL) r = r->rightChild;
      else r = r->leftChild;
      delete tmp;
    }
  }
  return r;
}
```

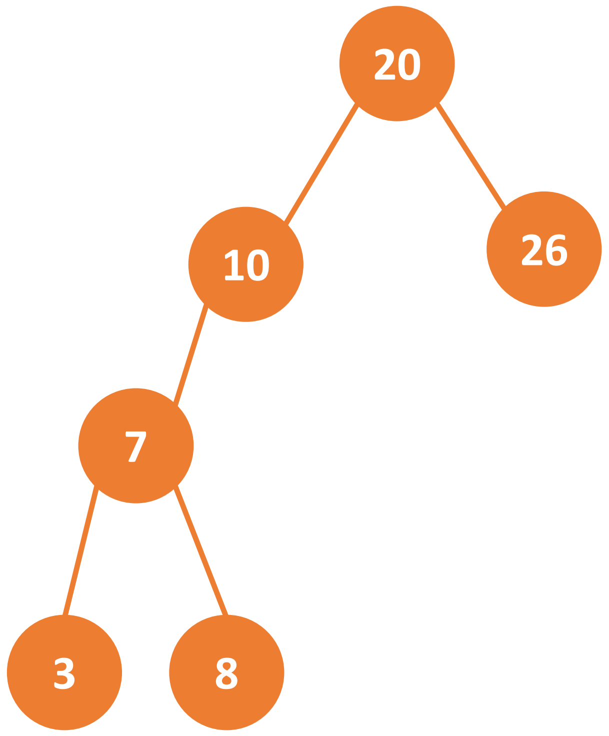
Balanced Binary Search Tree - AVL

- AVL is a BST with balance property
    - Difference of heights of the left child and the right child of each node is at most 1
    - The height of a AVL is log*N* (*N* is the number of nodes)
    - Insertion and removal of a node from the AVL must conserve the balance property

Balanced Binary Search Tree - AVL



AVL



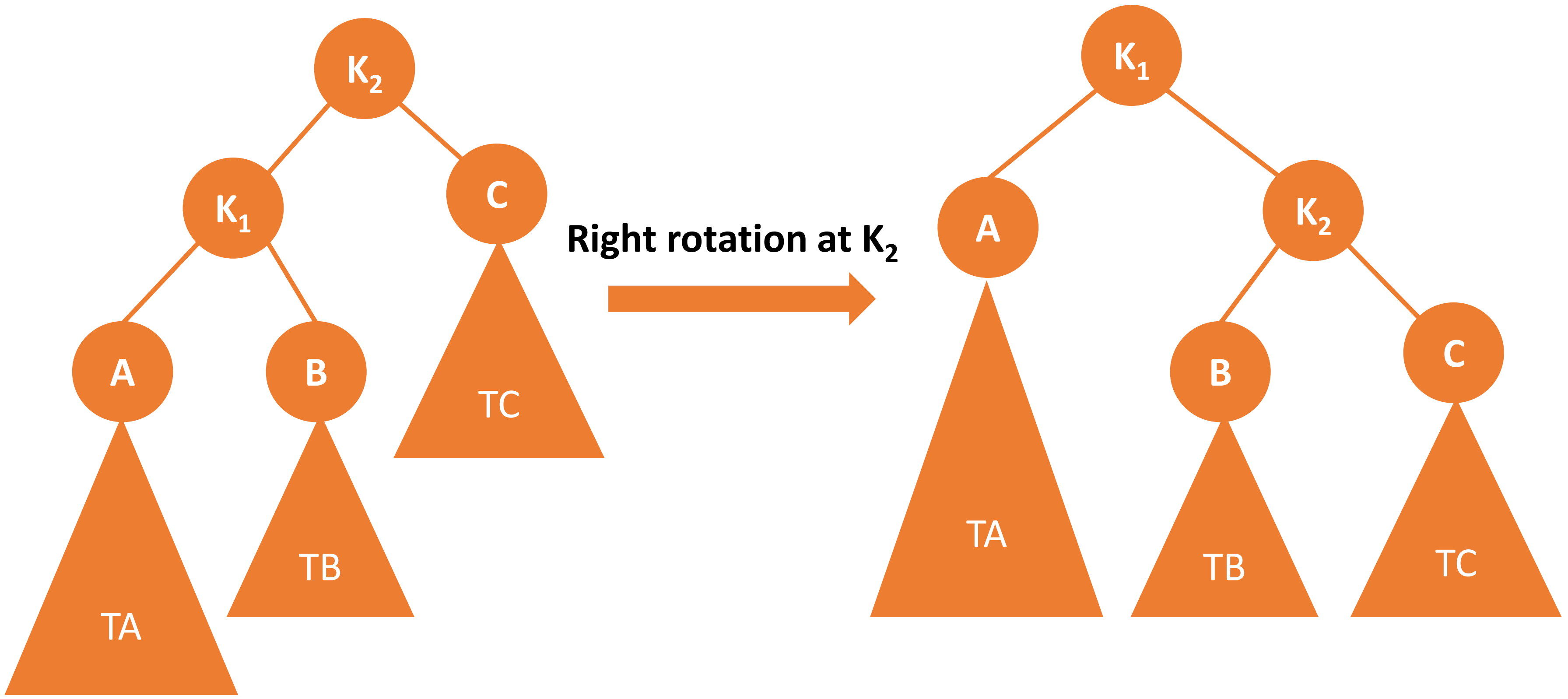
BST but not AVL

Balanced Binary Search Tree - AVL

- Difference of the heights of two children of a node might be 2 after the insertion or removal of a node
- Perform rotations to recover the balance property

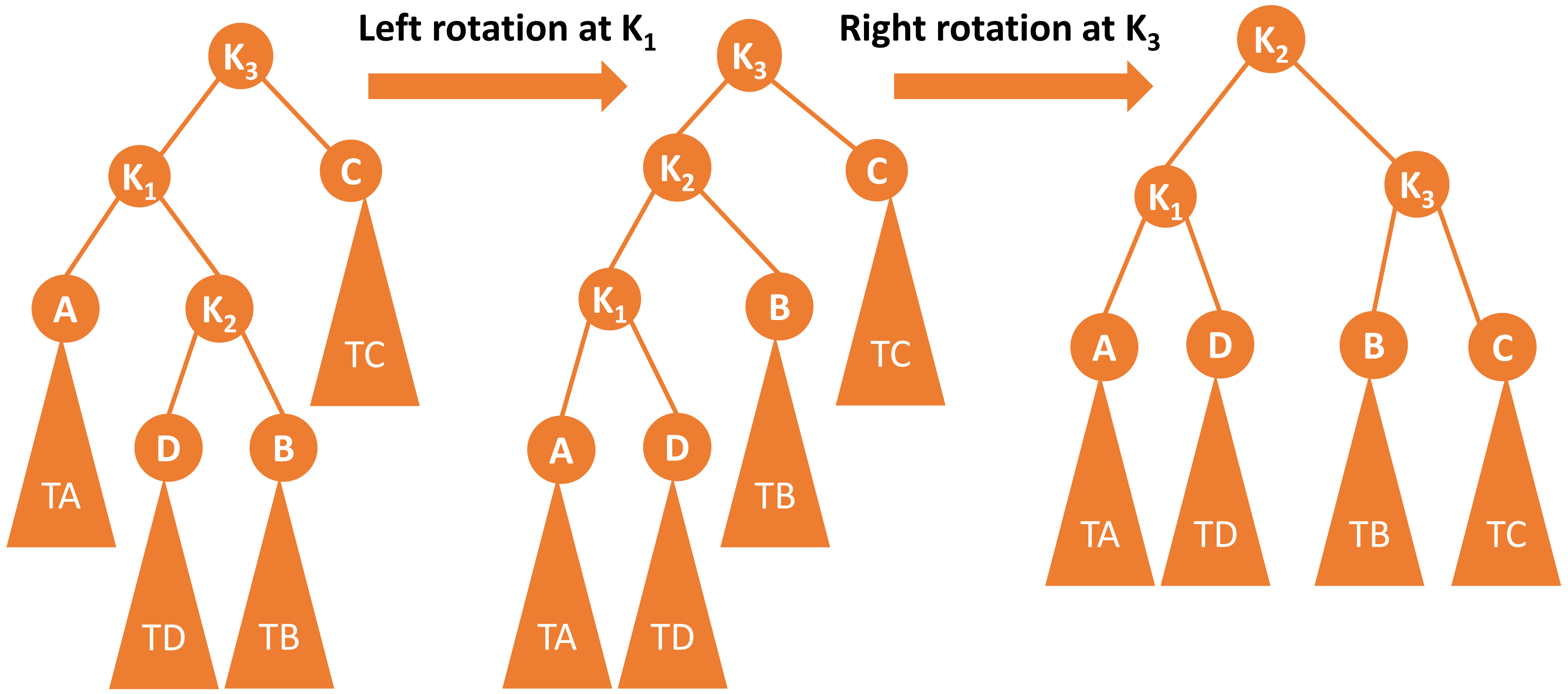
Balanced Binary Search Tree - AVL

Case 1



Balanced Binary Search Tree - AVL

Case 2

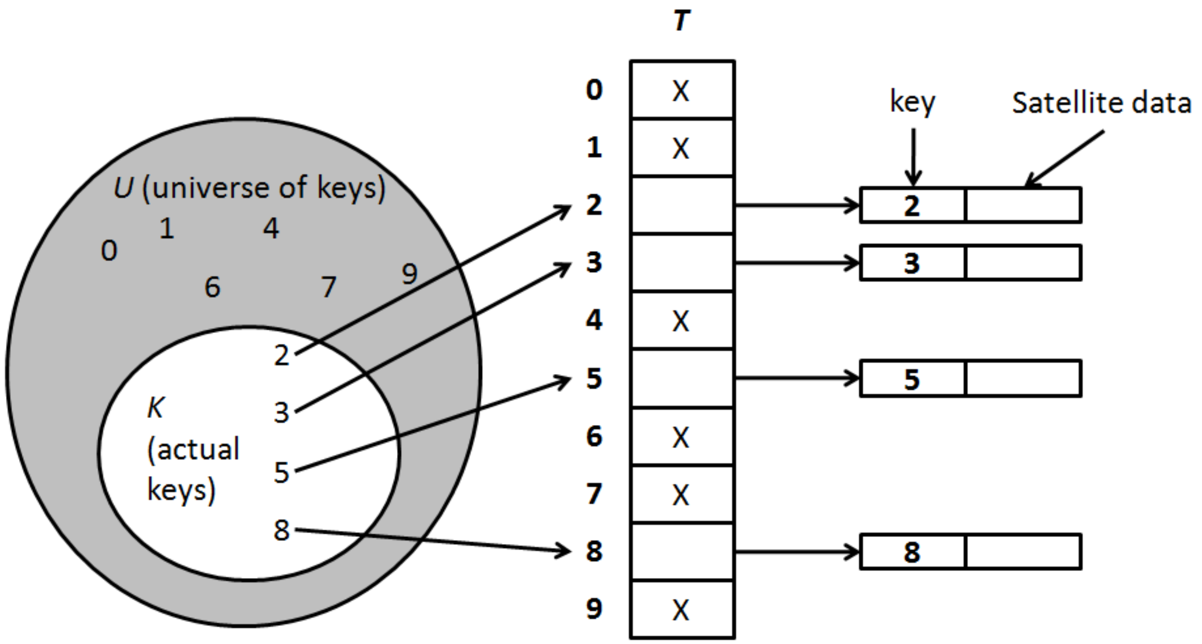


Hashing

- Mapping: a data structure storing pairs (key, value)
  - put(*k*,*v*): Set a mapping from a key *k* to a value *v*
  - get(*k*): return the value of the key *k*
- Implementation
  - Binary Search Trees
  - Hash tables

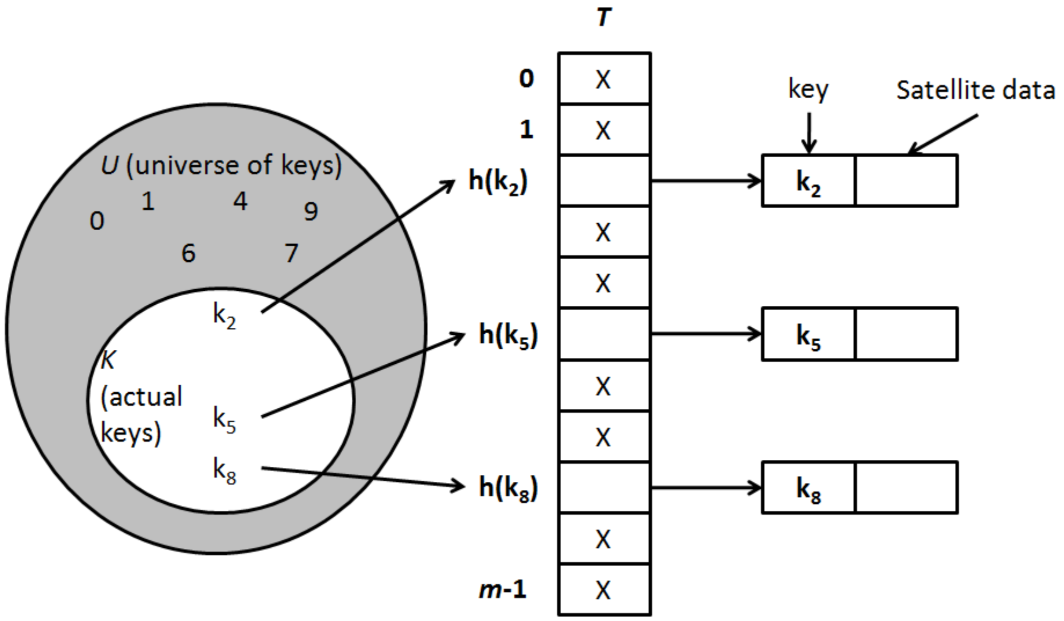
Hashing

- Direct addressing
  - Value of the key *k* is the address indicate the place in the table storing the pair (k,v)
  - Advantages: simple, fast lookup
  - Disadvantages: memory usage might be ineffective



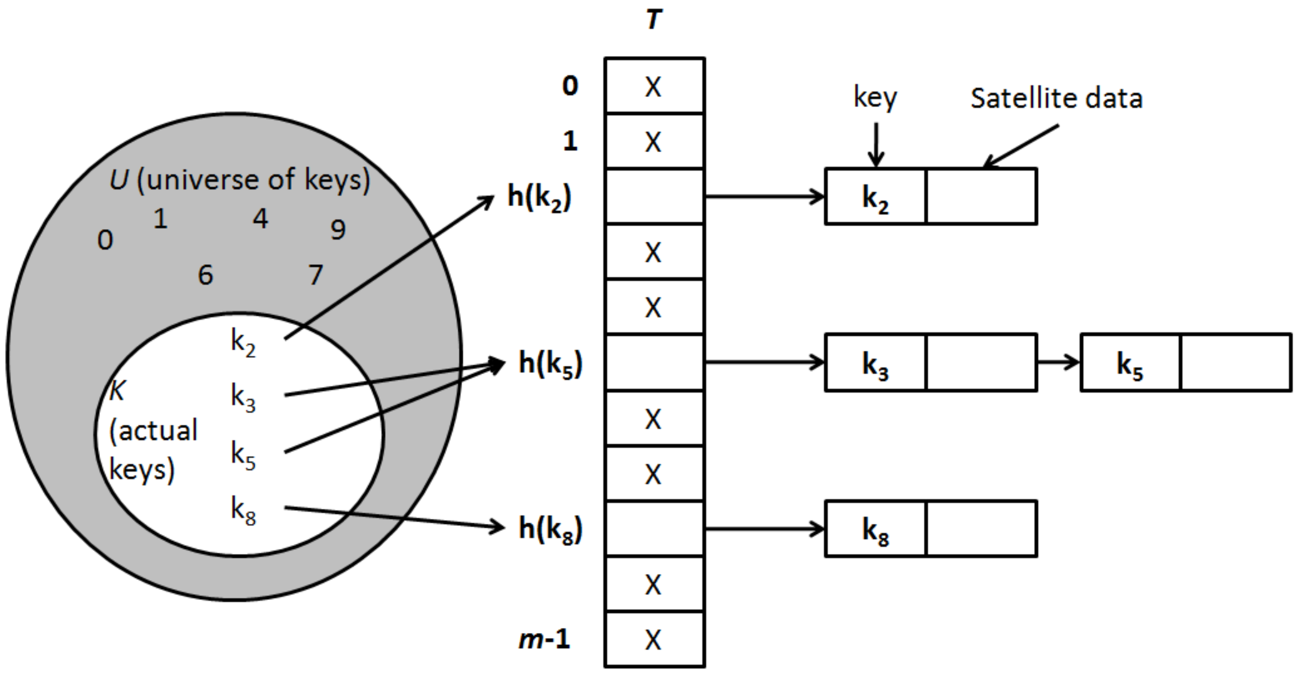
Hashing

- Hash function *h*(*k*) specifies the address where the pair (*k*, *value*) is stored
- *h*(*k*) should be simple and easy to compute



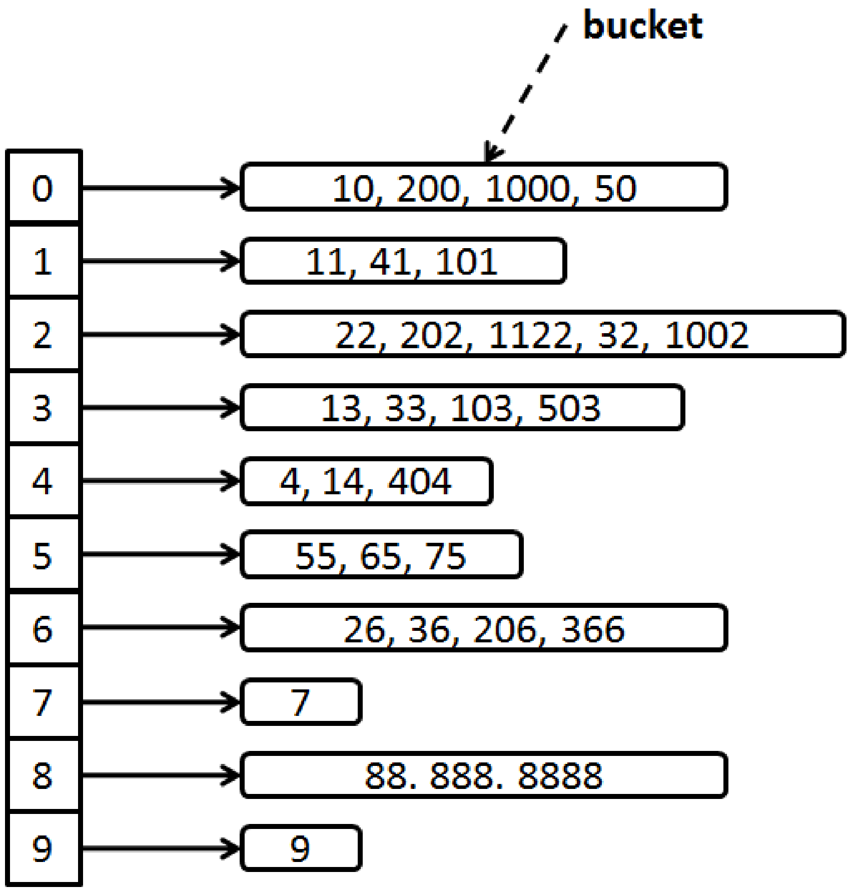
Hashing

- Collision: Two different keys have the same value of the hash function (hash code):
- Resolution:
  - Chaining: group keys having the same hash code into buckets
  - Open Addressing



Hashing

- Modulo: *h*(*k*) = *k* mod *m* where m is the size of the hash table



Hashing: Open Addressing

- Open Addressing
- Pairs (key, value) are stored in the table itself
- Operations put(k, v) and get(k) need to probe the table until the desired slot found
  - put(k, v): probe for finding a free slot for storing (k, v)
  - get(k): probe for finding the slot where the key k is stored
  - Probing order: $h(k, 0)$, $h(k, 1)$, $h(k, 2)$, …, $h(k, m\text{-}1)$
  - Methods
    - Linear probing: $h(k, i) = (h_1(k) + i) \bmod m$ where h_1 is normal hash function
    - Quadratic probing: $h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \bmod m$ where h_1 is normal hash function
    - Double hashing: $h(k, i) = (h_1(k) + i * h_2(k)) \bmod m$ where h_1 and h_2 are normal hash functions

Hashing: Open Addressing

```
get(k)
{
  // T: the table
  i = 0;
  while(i < m) {
    j = h(k,i);
    if(T[j].key = k) {
      return T[j];
    }
    i = i + 1;
  }
  return NULL;
}
```

```
put(k, v)
{
  // T: the table
  x.key = k; x.value = v;
  i = 0;
  while(i < m) {
    j = h(k,i);
    if(T[j] = NULL) {
      T[j] = x; return j;
    }
    i = i + 1;
  }
  error(“Hash table overflow”);
}
```

Hashing: Open Addressing

- Exercise: A table has *m* slots, apply the open addressing method in which $h(k, i)$ has the form:

$$h(k, i) = (k \bmod m + i) \bmod m$$

- Initialization, the table is free, present the status of the table after inserting following sequence of keys 7, 8, 6, 17, 4, 28 into the table with $m = 10$

Hashing: Open Addressing

- Exercise: A table has *m* slots, apply the open addressing method in which *h*(*k*, *i*) has the form:

$$h(k, i) = (k \bmod m + i) \bmod m$$

- Initialization, the table is free, present the status of the table after inserting following sequence of keys 7, 8, 6, 17, 4, 28 into the table with $m = 10$

0	1	2	3	4	5	6	7	8	9
28	x	x	x	4	x	6	7	8	17

Hashing functions

- Key is an integer
  - h(k) = mod m

- Key is a string
  - k = s[0..n] → h(k) = (s[0]*256^{n} + s[1]*256^{n-1} + … + s[n]*256^{0}) mod m

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

soict.hust.edu.vn/ fb.com/groups/soict