## Today's topics

- Prim's algorithm
- Hash table

## References

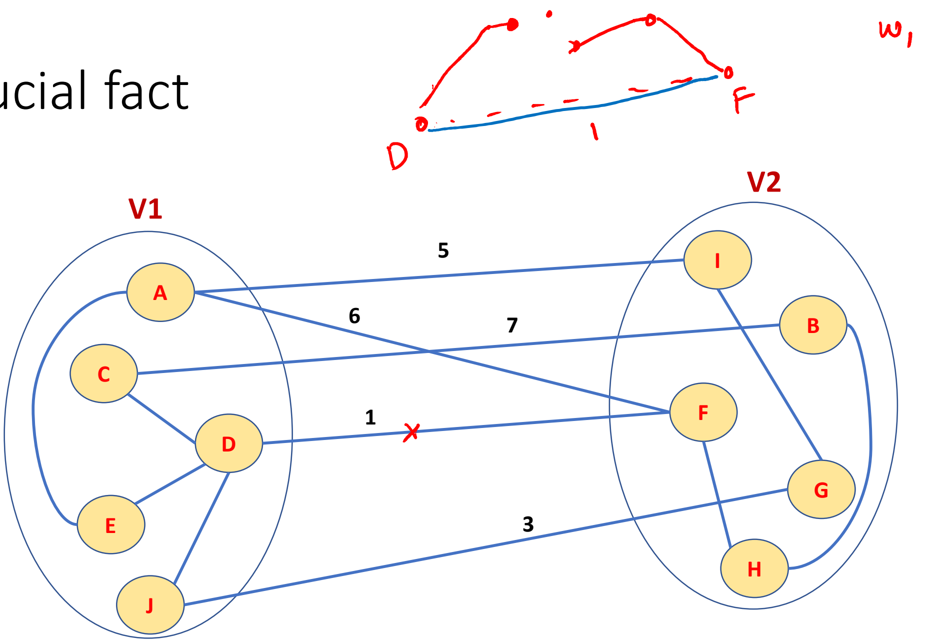
- Chapter-21.2 from the CLRS book
- Chapter-11 from the CLRS book
- Chapter-7.3 from Goodrich and Tamassia
- Chapter-5 from Mark Allen Weiss
- Chapter-2.5 from Goodrich and Tamassia

## A crucial fact

## A crucial fact

- Let G = (V, E, w) is a connected undirected graph, and $V_1$ and $V_2$ are two partitions of V such that $V_1 \cup V_2 = V \ and \ V_1 \cap V_2 = \phi$, the e be an edge in G with the minimum weight among all edges with one endpoint is in $V_1$ and the other endpoint is in $V_2$, then e must be in MST
  - This fact is the fundamental idea behind the algorithms for computing MST
- The disjoint-set partition of vertices is also called a cut

# A crucial fact



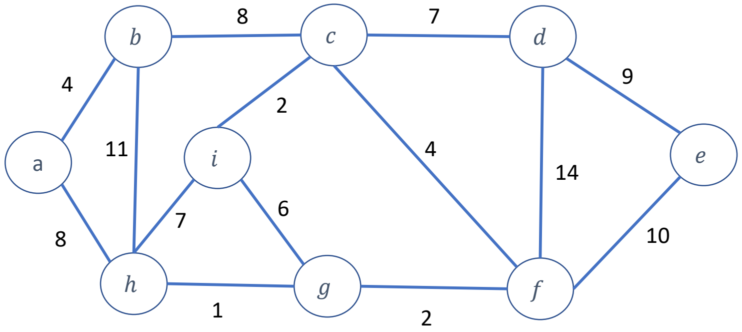
If we partition all vertices in the graph into two disjoint sets, V1 and V2, and S is the set of all edges between V1 and V2, the edge with the minimum weight in S must be part of the MST. In this example, edge (D, F) has the minimum weight. If this is not on the MST, then there must be an alternative path from D to F in the MST that contains at least one edge, say e, from the set S – {(D, F)}. Let's say T is the MST that doesn't contain (D, F). If we add edge (D, F) to T, the resulting graph must have the cycle containing both edges (D, F) and e. If we remove e from this cycle, the rest of the graph will still be connected and is a spanning tree. The weight of the new spanning tree is smaller than T because the weight of e is greater than that of (D, E). Thus T is not an MST.

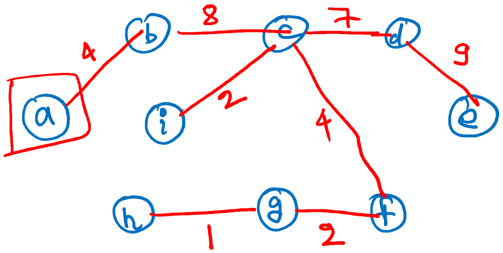
## A crucial fact

- Let’s say S is the set of all edges whose one endpoint is in V1 and the other endpoint is in V2, and edge e = (u, v) has the minimum weight among all edges in S
- Let’s say T is the spanning tree that doesn’t contain e
- Therefore, T must contain an edge $f \in S$ that is on the path from u to v
- If we add edge e to the graph, it’ll create a cycle containing edge f
- If we remove f from T and keep e, the resulting graph will still be the tree
- Because $w(e) <= w(f)$, if T was a spanning tree, the new tree is also a spanning tree

## Prim’s algorithm

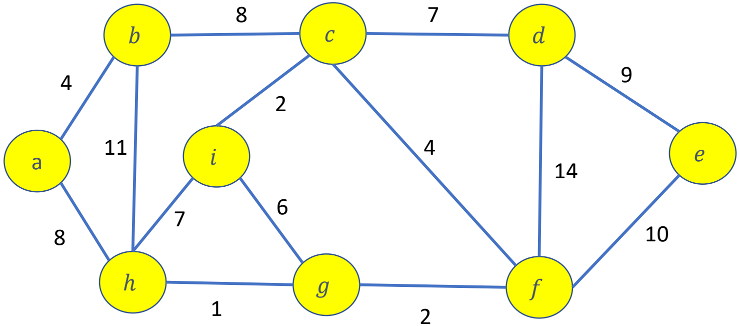
## Prim's algorithm





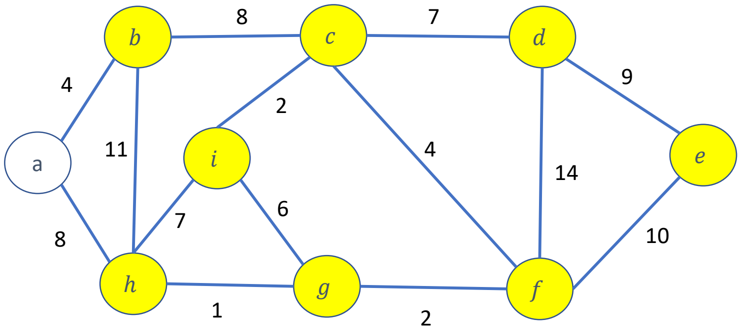
Prim's algorithm starts with a root vertex. It partitions the vertices into two sets: the first set contains just the root vertex, and the other set contains the remaining vertices. Now the algorithm picks the minimum edge from all the edges between the vertices of the first and second sets. The target vertex of the minimum edge is removed from the second set and added to the first set. Afterward, the same set of steps are executed on the first and second sets to move a vertex from the first set to the second set. This algorithm continues until all the vertices are added to the first set. Whenever a vertex is added to the first set, the corresponding minimum edge is part of the MST.

## Prim's algorithm



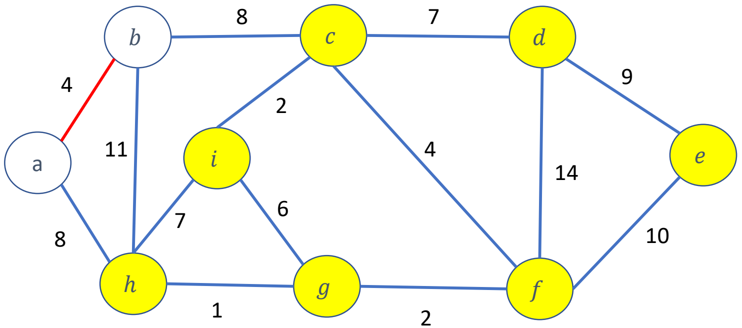
Initially, all vertices are in the second set.

## Prim’s algorithm



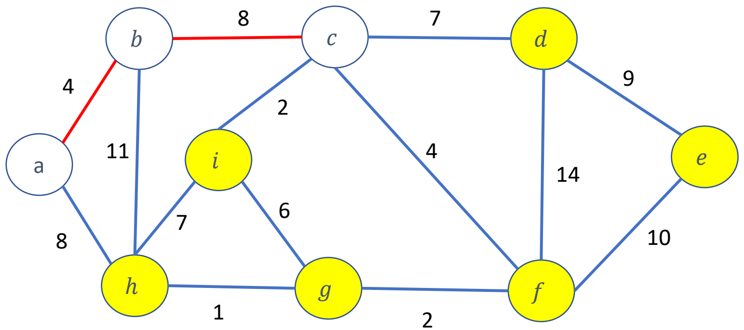
The root vertex a is added to the first set, and the rest of the vertices are in the second set.

## Prim’s algorithm



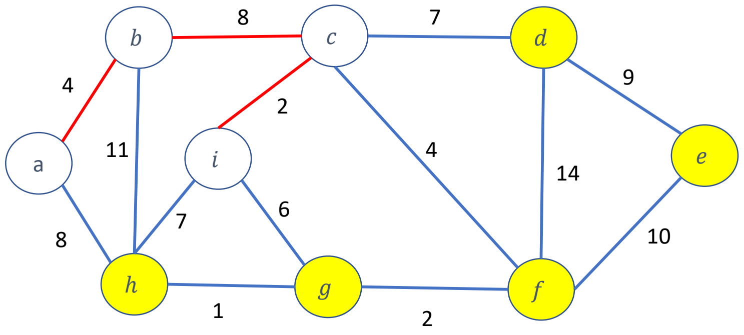
There are two edges, 4 and 8, going from the first set to the second set. We are adding the smaller edge, i.e., 4, to the MST and moving vertex b to the first set.

# Prim’s algorithm

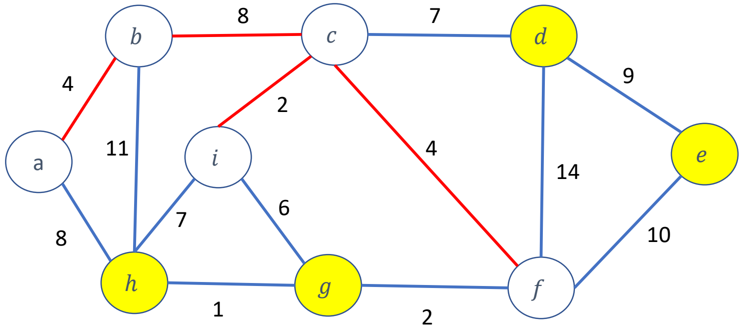


Now edges (a,h), (b.h), and (b,c) are going from the first set {a, b} to the second set {c, d, e, f, g, h, i}. We can pick either (b,c) or (a,h) because both are minimum. In this case, we pick (b,c); therefore, c is moved to the first set.

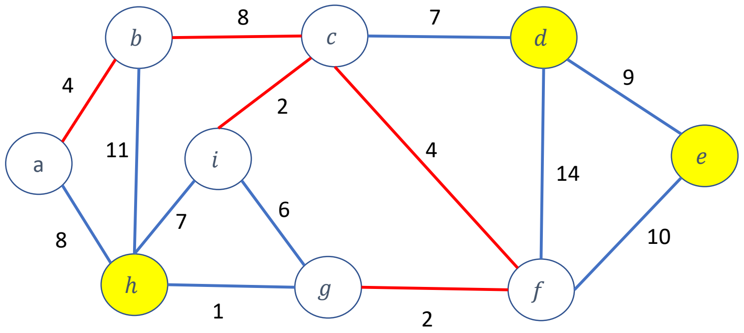
# Prim’s algorithm



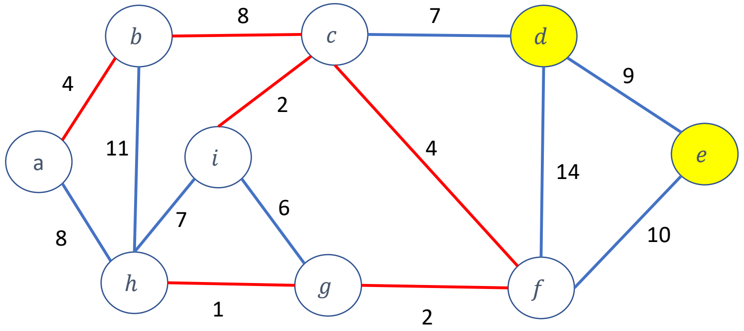
# Prim’s algorithm



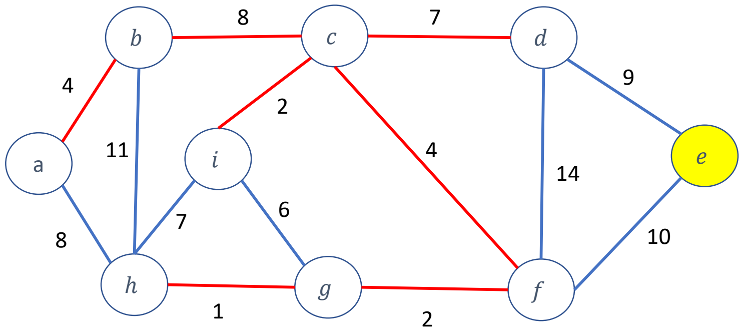
# Prim’s algorithm



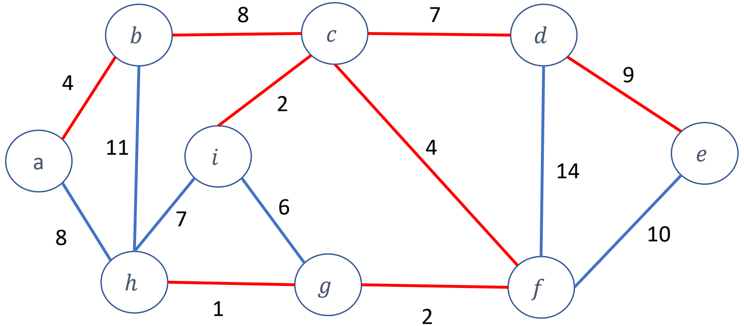
# Prim’s algorithm



# Prim's algorithm

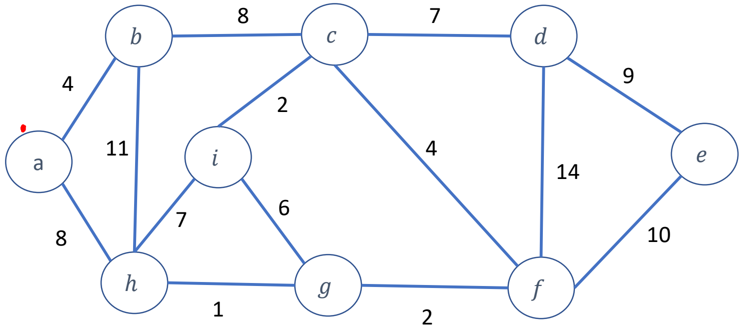


## Prim’s algorithm



## Implementation

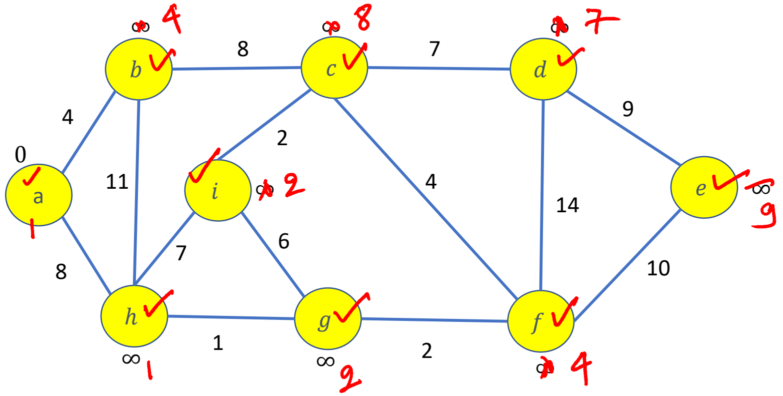
# Prim’s algorithm



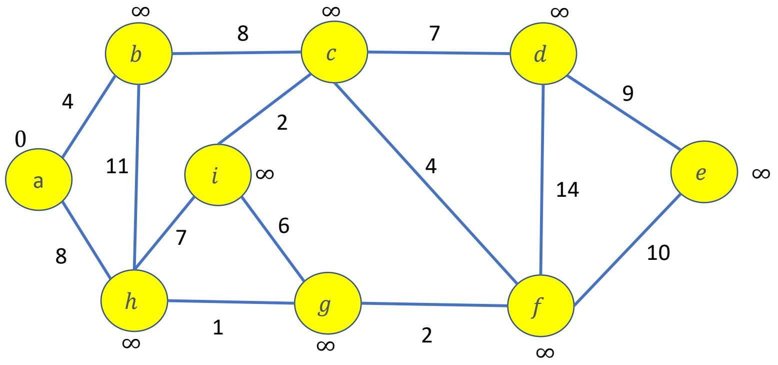
a f c
i b g

bh 11
ah 8
cd 7
ih 7
ig 6
fd 14
fe 10
gh 1

# Prim’s algorithm

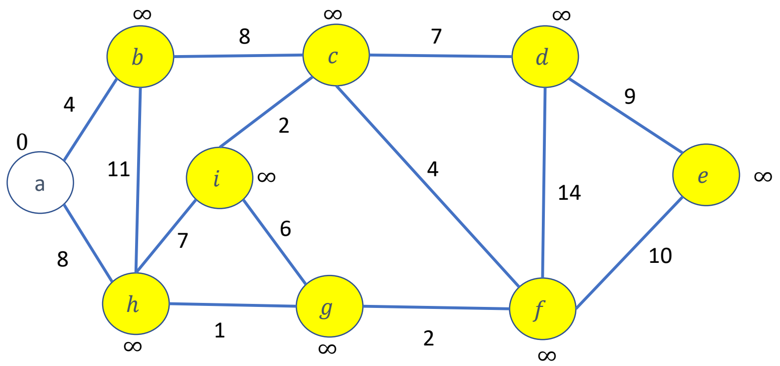


# Prim’s algorithm



```
v = Extract min
add v to spanning tree
update minimum edges
incident to other yellow
vertices adjacent to v
```

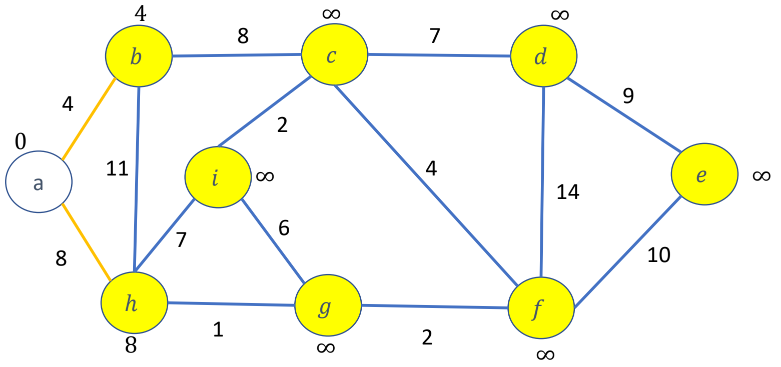
## Prim's algorithm



```
v = Extract min
add v to spanning tree
update minimum edges
incident to other yellow
vertices adjacent to v
```

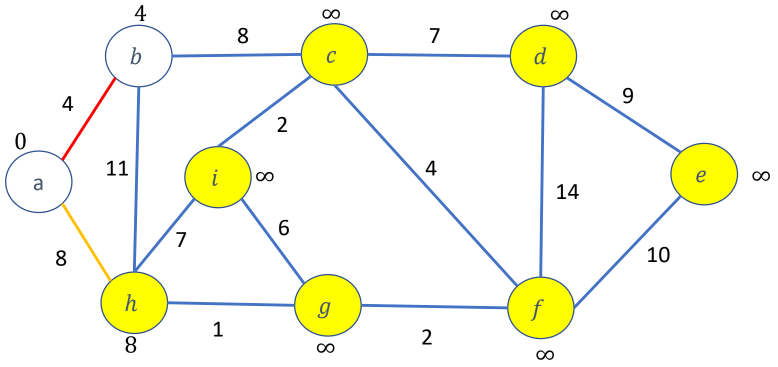
We can use an approach similar to Dijkstra. Each vertex contains a key field that stores the weight of the minimum weight edge among all edges incident to that vertex from the white vertices. Whenever a node becomes white, the keys of its yellow neighbors are updated if applicable.

# Prim’s algorithm



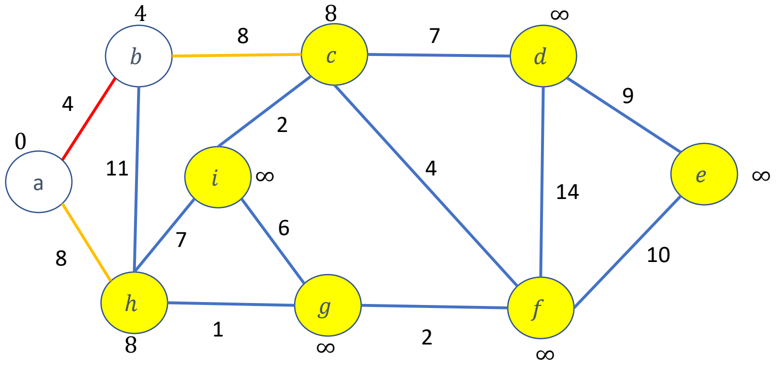
```
v = Extract min
add v to spanning tree
```
**update minimum edges**
**incident to other yellow**
**vertices adjacent to v**

## Prim’s algorithm



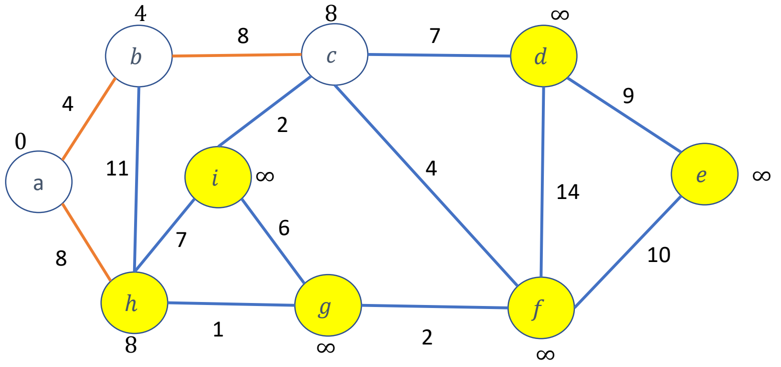
**v = Extract min**
**add v to spanning tree**
update minimum edges
incident to other yellow
vertices adjacent to v

# Prim's algorithm



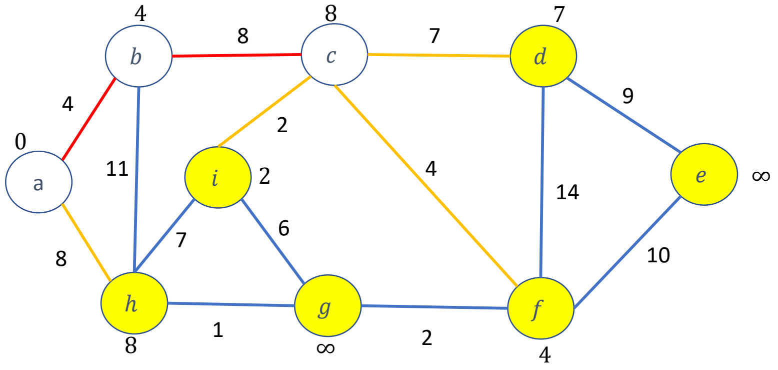
```
v = Extract min
add v to spanning tree
update minimum edges
incident to other yellow
vertices adjacent to v
```

# Prim’s algorithm



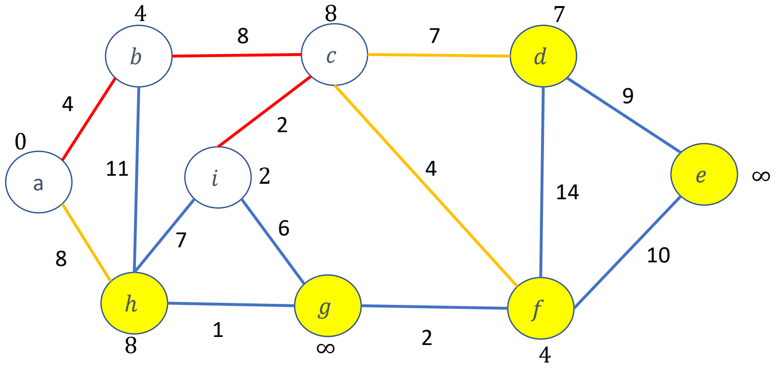
```
v = Extract min
add v to spanning tree
update minimum edges
incident to other yellow
vertices adjacent to v
```

# Prim’s algorithm



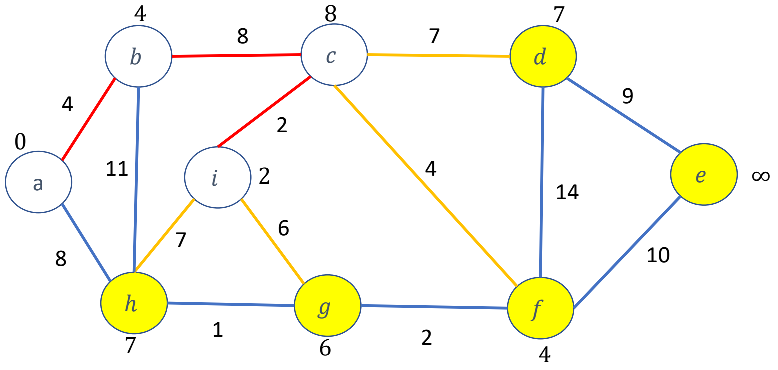
```
v = Extract min
add v to spanning tree
```
**update minimum edges incident to other yellow vertices adjacent to v**

## Prim’s algorithm



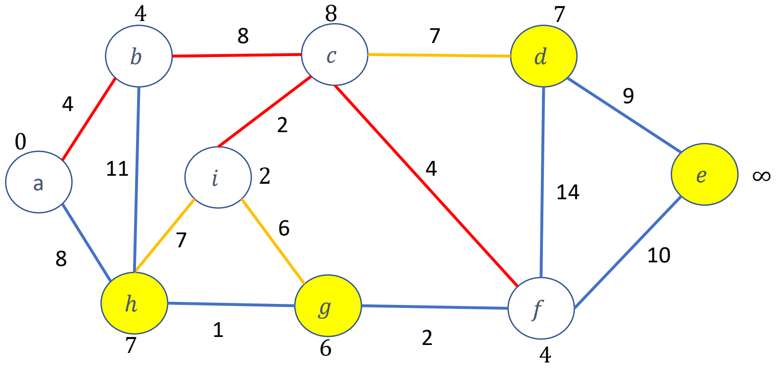
```
v = Extract min
add v to spanning tree
update minimum edges
incident to other yellow
vertices adjacent to v
```

# Prim’s algorithm



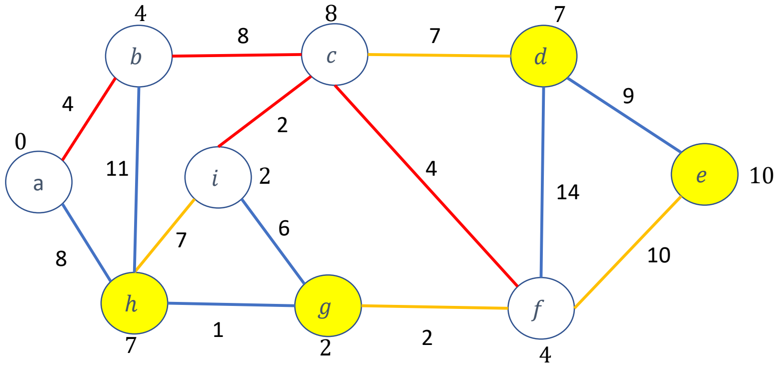
```
v = Extract min
add v to spanning tree
```
**update minimum edges**
**incident to other yellow**
**vertices adjacent to v**

## Prim’s algorithm



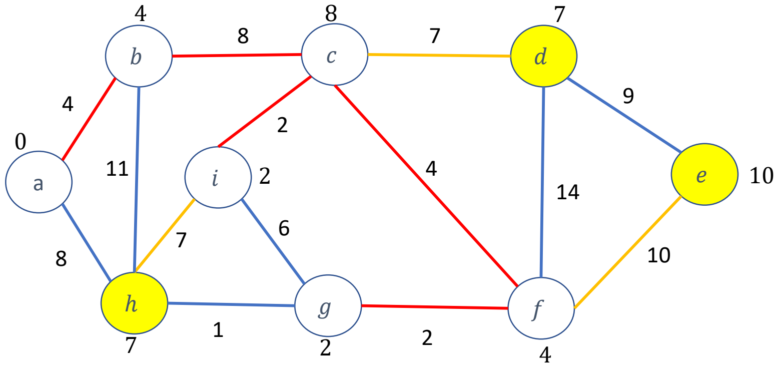
```
v = Extract min
add v to spanning tree
update minimum edges
incident to other yellow
vertices adjacent to v
```

# Prim’s algorithm



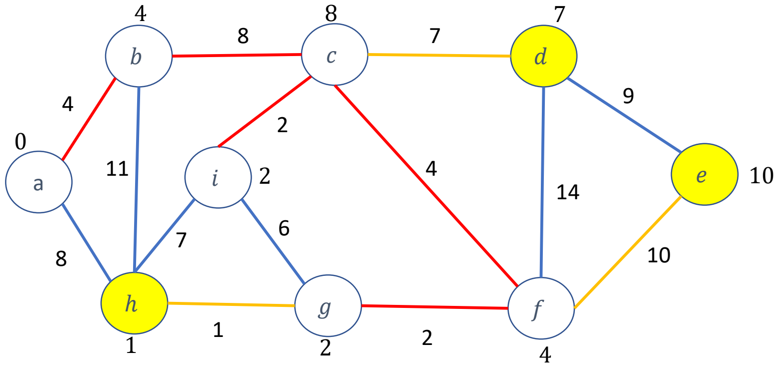
```
v = Extract min
add v to spanning tree
```
**update minimum edges incident to other yellow vertices adjacent to v**

# Prim’s algorithm



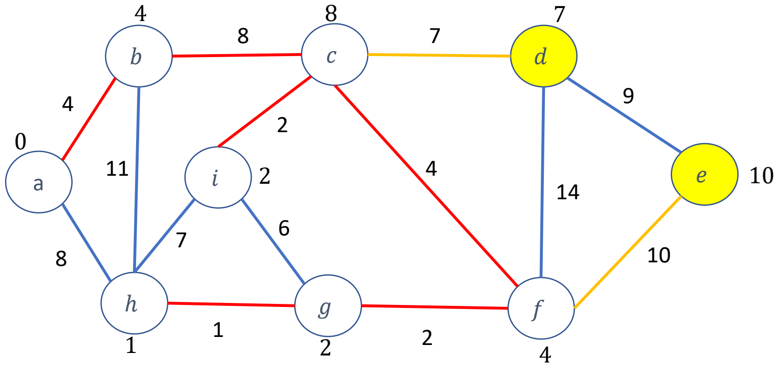
**v = Extract min**
**add v to spanning tree**
update minimum edges
incident to other yellow
vertices adjacent to v

# Prim’s algorithm



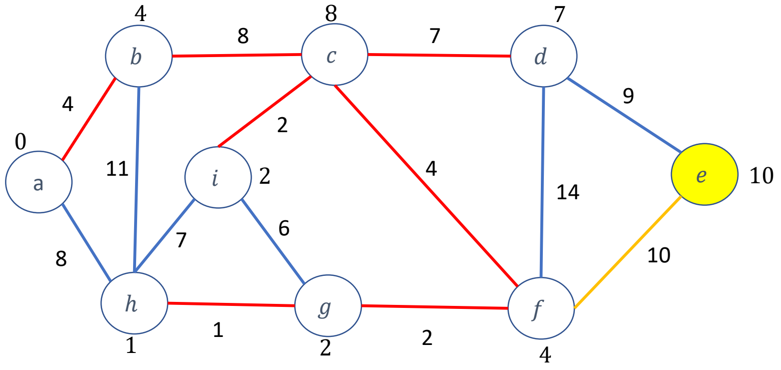
```
v = Extract min
add v to spanning tree
```
```
**update minimum edges
incident to other yellow
vertices adjacent to v**
```

# Prim's algorithm



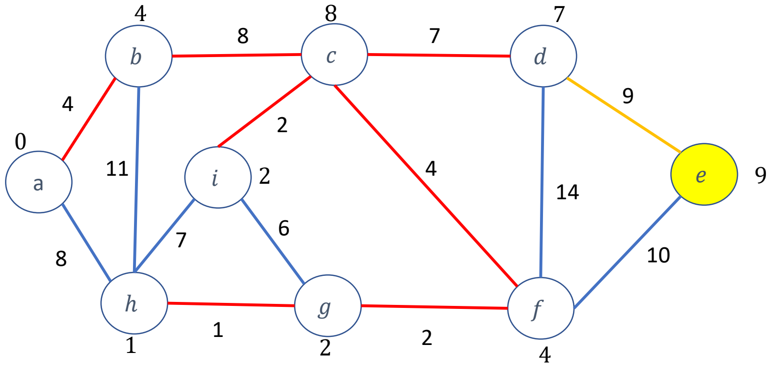
```
v = Extract min
add v to spanning tree
update minimum edges
incident to other yellow
vertices adjacent to v
```

# Prim’s algorithm



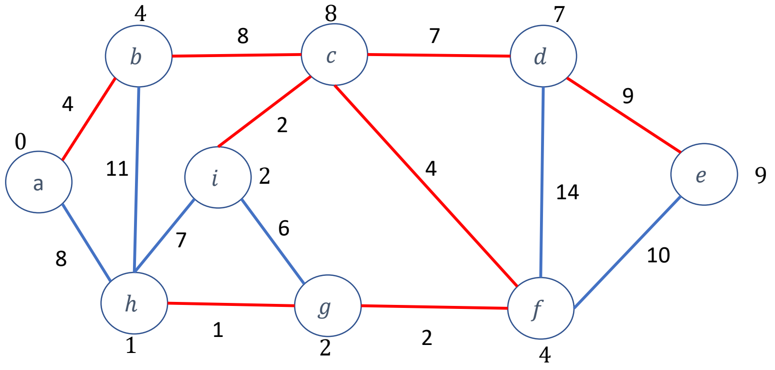
```
v = Extract min
add v to spanning tree
update minimum edges
incident to other yellow
vertices adjacent to v
```

# Prim’s algorithm



```
v = Extract min
add v to spanning tree
```
```
**update minimum edges
incident to other yellow
vertices adjacent to v**
```

# Prim's algorithm



**v = Extract min**
**add v to spanning tree**
update minimum edges
incident to other yellow
vertices adjacent to v

## Implementation

- We can implement Prim's algorithm in the similar way we implemented the Dijkstra algorithm
- The vertices that are added to the spanning tree are marked as white
- Other vertices are marked as yellow
- For yellow vertices, we maintain the minimum edge that is coming from the set of white vertices
- At each step, we find the vertex that is at the shortest distance from the set of white vertices and make it white
- After a yellow vertex, x, becomes a white vertex, the minimum distances from the white vertices may change for all yellow vertices adjacent to x. If applicable, we update the minimum distance of these vertices

## Prim’s algorithm

$|E| \log(|V|)$

```
MST_PRIM(G, r)
// G is the graph G = (V, E, w)
// r is the root node
// each vertex v has two fields key and $\pi$
// Output: a set that contains all edges in the MST

for each vertex $u \in G.V$
    u.key = ∞
    u.$\pi$ = NIL
r.key = 0
Q = $\phi$
for each vertex u in G.V
    INSERT(Q, u)
while Q != $\phi$
    u = EXTRACT_MIN(Q)
for each vertex v in G.Adj[u]
    if $v \in Q$ and w(u, v) < v.key
        v.$\pi$ = u
        v.key = w(u, v)
        DECREASE_KEY(Q, v, w(u,v))
```

A vertex contains two fields: key and $\pi$. key is the weight of the minimum edge incident to the vertex from white vertices. $\pi$ contains the other endpoint of the edge whose weight is stored in key.

## Prim’s algorithm

- Time complexity

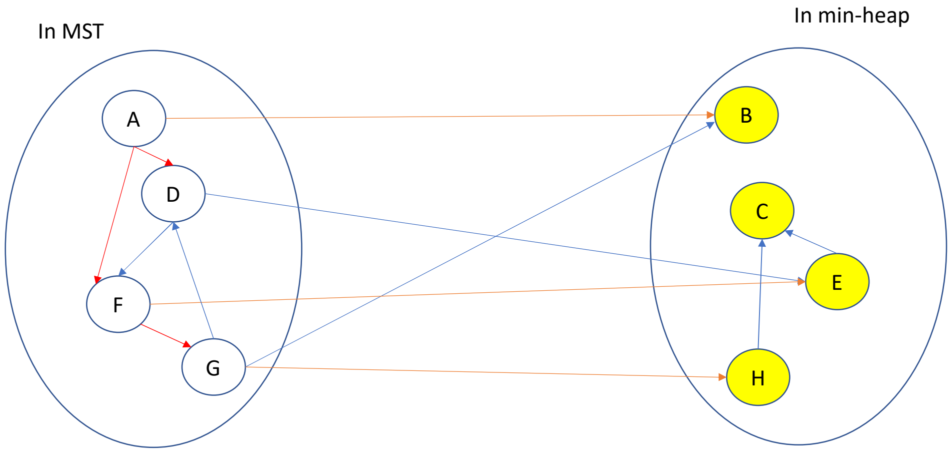
```
1. MST_PRIM(G, r)
2. // G is the graph G = (V, E, w)
3. // r is the root node
4. // each vertex v has two fields key
   and $\pi$
5. // Output: a set that contains all
   edges in the MST; for each vertex v,
   $\pi$ contains the predecessor of v in
   the MST

6. for each vertex $u \in G.V$
7.    u.key = ∞
8.    u.$\pi$ = NIL
9. r.key = 0
10.Q = $\phi$
11.for each vertex u in G.V
12.   INSERT(Q, u)
13.while Q != $\phi$
14.   u = EXTRACT_MIN(Q)
15.   for each vertex v in G.Adj[u]
16.      if $v \in Q$ and w(u, v) < v.key
17.         v.$\pi$ = u
18.         v.key = w(u, v)
19.         DECREASE_KEY(Q, v, w(u,v))
```

## Time complexity

- Line-11,12 takes O(|V|) operations if implemented using BUILD_MIN_HEAP
- Line-14 executes |V| times and thus takes O(|V|*log(|V|)) operation
- Line-19 may execute |E| times and thus the number of operations take O(|E| * log(|V|)) time
- Therefore, the overall time complexity is O(|E| * log(|V|))
    - Notice the graph is connected and therefore |E| >= |V|-1

# Prim’s algorithm invariants



**Invariant-1 :** The white vertices are already in the MST.
**Invariant-2:** For a yellow vertex v, if $v \neq \infty$ and $v.\pi \neq NIL$, then v.key is equal to the weight of the minimum weight edge out of all the edges incident to v from the set of white vertices.
**Can you identify any similarity with the Dijkstra algorithm?**

## Prim’s algorithm invariants

- The prim’s algorithm maintains the following invariant before every iteration in the while loop
    1. The vertices in the set V – Q are already in the MST
    2. All vertices in v ∈ Q, if $v \neq \infty$ and $v.\pi \ \neq NIL$, then v.key is equal to the weight of the minimum weight edge out of all the edges incident to v from the set of vertices in V-Q

## Prim’s algorithm invariants

- The prim’s algorithm maintains the following invariant before every iteration in the while loop
    1. The vertices in the set V – Q are already in the MST
    2. All vertices in v ∈ Q, if $v \neq \infty$ and $v.\pi \ \neq NIL$, then v.key is equal to the weight of the minimum weight edge out of all the edges incident to v from the set of vertices in V-Q

  - Does invariants-1,2 hold after removing a vertex u for Q and adding it to MST?
    - Invariant-1 holds for all the vertices in Q that are not adjacent to u
    - Invariant-1 also holds for all the vertices v ∈ Q adjacent to u because if (u, v) is shorter than incident edges on v from V – Q – {u}, we are setting v.key to the w(u, v) at line-19

## Prim’s algorithm invariants

- The prim’s algorithm maintains the following invariant before every iteration in the while loop
  1. The vertices in the set V – Q are already in the MST
  2. All vertices in v ∈ Q, if $v \neq \infty$ and $v.\pi \ \neq NIL$, then v.key is equal to the weight of the minimum weight edge out of all the edges incident to v from the set of vertices in V-Q

  - Does invariants-1,2 hold after removing a vertex u for Q and adding it to MST?
    - Because of invariant-2, we know that for every vertex $v \in Q$, v.key and v.pred represent the minimum weight edge incident to v from V – Q. Because of the crucial fact about MST, we know that the minimum edge among all edges between V – Q and Q is in the MST. The edge that is moved for Q to MST at line-14 has the minimum key among all vertices in Q. Therefore, it is indeed the minimum edge incident to Q from V-Q, and thus invariant-1 also holds.

## Hash table

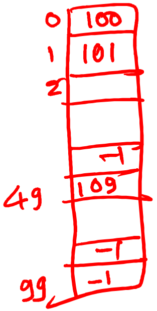
## Hash table

- Let’s say we have the following key-value pairs. All values are positive. How can we efficiently store these key-value pairs in a data-structure such that finding the value corresponding to a key is cheaper? The maximum value of a key is 100. There are no duplicate keys.

[0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109]

## Hash table

[0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109]



We can create an array of size 100 and store the value at arr[key].

## Hash table

- Let’s say we have the following key-value pairs. All values are positive. How can we efficiently store these key-value pairs in a data-structure such that finding the value corresponding to a key is cheaper? The maximum number of key-value pairs can be 100. There are no duplicate keys.

[0, 100], [1, 101], [1081, 102], [40004, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109]

## Hash table

[0, 100], [1, 101], [1081, 102], [40004, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109]

In this case, we need to use a large array because the key is very large. Another alternative is to store the value at (key%100). However, in this case, we need to store both key and value, because the array index is no longer the key.

## Hash table

- Let’s say we have the following key-value pairs. All values are positive. How can we efficiently store these key-value pairs in a data-structure such that finding the value corresponding to a key is cheaper? The maximum number of key-value pairs can be 100. There are no duplicate keys.

[“Amar”, 100], [“Akbar”, 101], [“Anthony”, 102], [“Carlos”, 103], [“Robert”, 104], [“Pran”, 105], [“Amrish”, 106], [“Amjad”, 107], [“Danny”, 108], [“Shakti”, 109]

# Hash table

[“Amar”, 100], [“Akbar”, 101], [“Anthony”, 102], [“Carlos”, 103], [“Robert”, 104], [“Pran”, 105], [“Amrish”, 106], [“Amjad”, 107], [“Danny”, 108], [“Shakti”, 109]

Amar $\rightarrow$ $\left(\text{ASCII}(A) + \text{ASCII}(m) + \text{ASCII}(a) + \text{ASCII}(r)\right) \% 100$

We can use the sum of the ASCII values of all characters as an index in the array.

## Hash table

- The hash table is a key value store
- The primary use case of a hash table is to find the value associated with a key if the key is present in the hash table
- The hash table operations are: insert, delete, and find

## Hash table

- Hash table leverages the fact that the array indexing is fast
- We can store the value at an index equal to the associated key
- To find the value associated with a key, we can directly look at the corresponding index

## Hash table

- There are several challenges in implementing a hash table
  - If the key is equal to the index, then the key must be an integer
  - If the number of elements in the hash table is small, but the values of some keys are large, then the size of the array would be very large

- To address these problems, we can map the key to an integer value lower than the size of the hash table; the new value is used to index in the array

- The function that is used to map the key to an integer value is called a hash function

## Hash function

- If the key is a string, we can convert it into an integer by adding the ASCII values of all characters in the string
- We can also use an equation that uses the ASCII values of the characters and their positions in the strings to generate an integer value
- For integer keys, we can use key % m, where m is the size of the hash table
- You can define the hash function in whatever ways you like as long as it maps the key to a valid index in the hash table

## Hash function

- One problem with the hash functions is that two keys can map to a single index in the hash table even if the hash table is large enough to store all (key, value) pairs

- This event is called a hash collision

- A good hash function tries to minimize the hash collisions, but completely avoiding hash collisions for arbitrary data may not be possible

## Hash function

- Therefore, the hash table must implement a strategy to deal with the hash collisions

- The first strategy that is used is called chaining

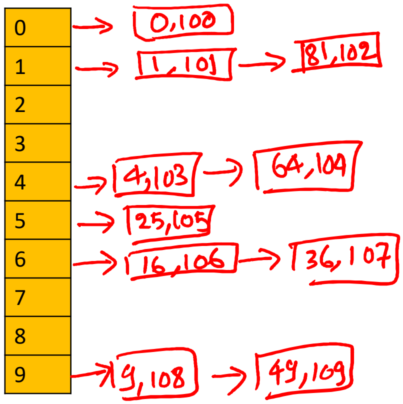
## Chaining

## Chaining

- In the chaining strategy, keys with the same hash values are stored in a linked list
- The hash table maintains m linked-lists, one for every slot, where m is the number of slots in the hash table
- During insertion, the key-value pair is inserted in the linked-list at an index equal to the hash value of the key
- To find an element corresponding to a key, we need to iterate the linked list stored at the index equal to the hash value of key

# Chaining

Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.

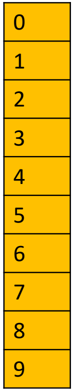


```
int HashFun(int x) {
    return x % 10;
}
```

Each slot in the hash table points to a linked-list containing all key-value pairs mapped to that slot.

## Chaining

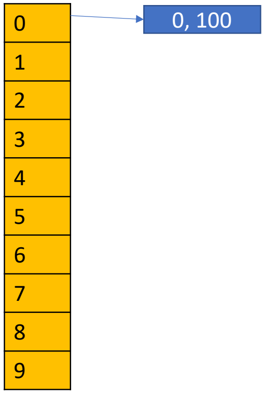
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

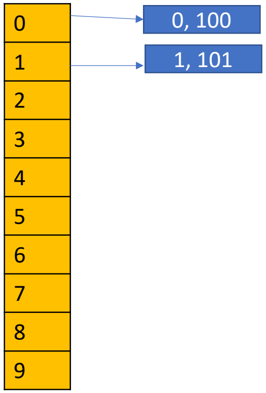
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

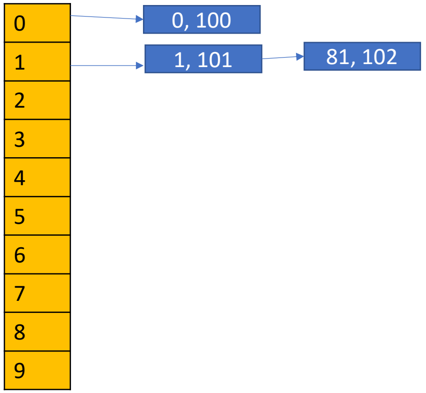
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

# Chaining

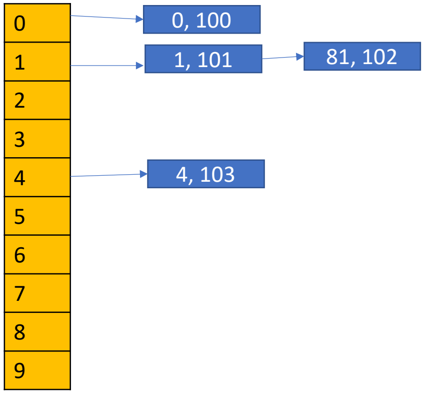
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

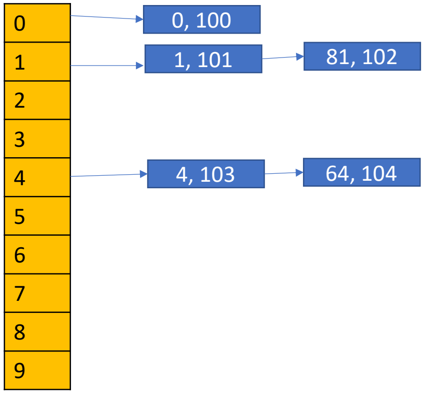
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

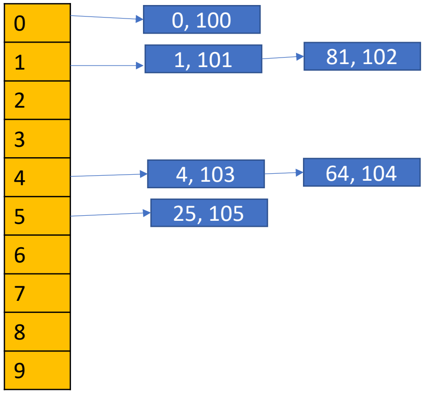
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

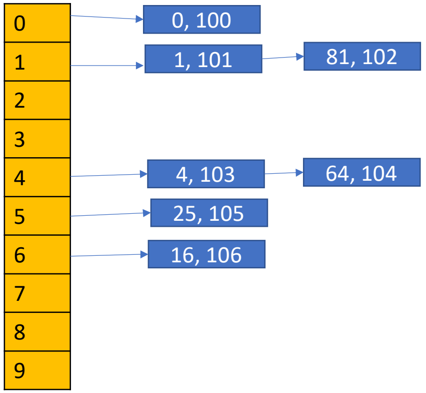
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

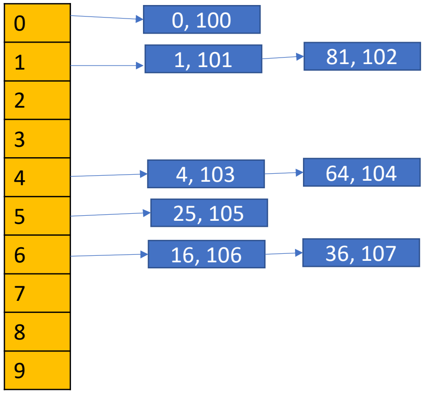
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

# Chaining

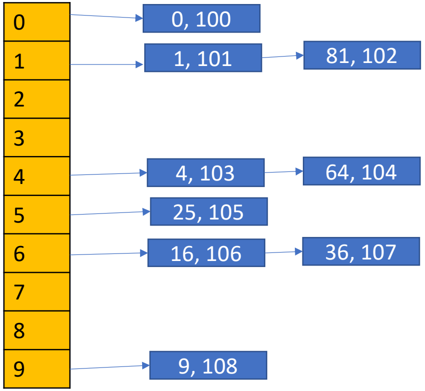
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

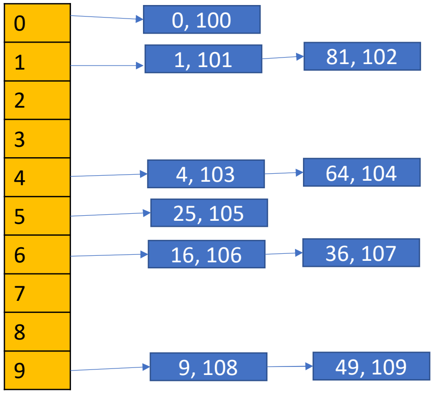
Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

Insert [0, 100], [1, 101], [81, 102], [4, 103], [64, 104], [25, 105], [16, 106], [36, 107], [9, 108], [49, 109] into a hash table with chaining.



```
int HashFun(int x) {
    return x % 10;
}
```

## Chaining

- Time complexity (assuming hash function is O(1))
  - Insert $O(1)$
  - Find $O(n)$
  - Delete $O(n)$

## Hash function

- The efficiency of the hash tables depends upon how good the hash function is

- A good hash function uniformly distributes the key, i.e., each key is equally likely to hash to any of the m slots, where m is the number of slots in the hash table

## Hash function

- The division method

$h(k) = k \bmod m$, where $m$ is the number of slots in the hash table

## Hash function

- The division method

$h(k) = k \bmod m$, where $m$ is the number of slots in the hash table

In this hash function, the choice of m is critical:

If $m = 2^p$, $h(k)$ is lowest p bits of k

If the lowest p bits are not equally likely, then this might not be a good hash function

If m is 10 and most keys end in zero, then it’s not a good choice

A prime number is considered a good choice for m

## Hash function

- What is the problem with the following hash function?

```
// key is a null terminated character array
// m is the number of slots in the hash table
int HashFn(char *key, int m)
{
  unsigned int HashVal = 0;
  int i;
  for (i = 0; key[i] != ‘\0’; i++)
    HashVal += key[i];
  return HashVal % m;
}
```

## Hash function

- What is the problem with the following hash function?

```
// key is a null terminated character array
// m is the number of slots in the hash table
int HashFn(char *key, int m)
{
  unsigned int HashVal = 0;
  int i;
  for (i = 0; key[i] != ‘\0’; i++)
    HashVal += key[i];
  return HashVal % m;
}
```

Because the ASCII value of a character is less than 128, the calculated HashVal could be much lower than m. Also, two different strings with the same characters will be mapped to the same index in the hash table.

## Hash function

- What is the problem of the following hash function?

```
// key is a null terminated character array
// m is the number of slots in the hash table
// this code assumes that the string has at least three characters
int HashFn(char *key, int m)
{
  return (key[0] + (27 * key[1]) + (729 * key[2])) % m;
}
```

## Hash function

- What is the problem of the following hash function?

```
// key is a null terminated character array
// m is the number of slots in the hash table
// this code assumes that the string has at least three characters
int HashFn(char *key, int m)
{
  return (key[0] + (27 * key[1]) + (729 * key[2])) % m;
}
```

```
Even though there are 26 * 26 * 26 = 17,276 different possible strings
of length three, if the keys are valid English words, then the possible
valid strings are roughly 2,851. If m is around 10000, only 28% of the
table will be occupied even if there are no hash collisions.
```

## Hash function

- A good hash function.

```
// key is a null terminated character array
// m is the number of slots in the hash table
int HashFn(char *key, int m) {
  unsigned int HashVal = 0;
  int i;
  for (i = 0; key[i] != ‘\0’; i++)
    HashVal = (HashVal << 5) + key[i];
  return HashVal % m;
}
```

This function computes: $\sum_{i=0}^{KeySize-} key[KeySize - i - 1] * 32^i$

## Hash function

- A good hash function.

```
// key is a null terminated character array
// m is the number of slots in the hash table
int HashFn(char *key, int m) {
  unsigned int HashVal = 0;
  int i;
  for (i = 0; key[i] != ‘\0’; i++)
    HashVal = (HashVal << 5) + key[i];
  return HashVal % m;
}
```

This function computes: $\sum_{i=0}^{KeySize-1} key[KeySize - i - 1] * 32^i$

It computes a polynomial function on all characters and their positions in the string. The computed result is expected to distribute well. This hash function is also efficient.

# Hash function

- A good hash function.

```
// key is a null terminated character array
// m is the number of slots in the hash table
int HashFn(char *key, int m) {
  unsigned int HashVal = 0;
  int i;
  for (i = 0; key[i] != '\0'; i++)
    HashVal = (HashVal << 5) + key[i];
  return HashVal % m;
}
```

This function computes: $\sum_{i=0}^{KeySize-1} key[KeySize-i-1] * 32^{i}$

However, this hash function might take a long time if the length of the string is large. In addition, the earlier characters may not contribute to the hash value at all because of the overflow. A common practice is to not use all characters. The length and properties of the key are crucial in designing the hash function for a given workload.

## Chaining

- The average number of elements stored in a linked-list is also called the load factor

- A uniform hash function returns any of the m slots with equal probability for an input key

- What is the load factor of a hash table with m slots and n entries if a uniform hash function is used?

## Chaining

- The average number of elements stored in a linked-list is also called the load factor

- A uniform hash function returns any of the m slots with equal probability for an input key

- What is the load factor of a hash table with m slots and n entries if a uniform hash function is used?
  - n/m

## Chaining

- What is the average time complexity of the search and delete operations in a hash table with chaining?

## Chaining

- What is the average time complexity of the search and delete operations in a hash table with chaining?
  - O(load factor)

## Open addressing

## Open addressing

- In open addressing, all elements are stored in hash table slots only
  - No additional linked list is created

- Linear probing and quadratic probing are some of the techniques that are used to handle hash collisions

## Linear probing

## Linear probing

- In the linear probing scheme, if slot $i$ returned by the hash function is already occupied
  - the element is inserted into the first available slot between $i+1$ to $m-1$, where $m$ is the number of slots in the hash table
  - If none of the slots is available between $i+1$ to $m-1$, the element is inserted at the first available slot between $0$ to $i-1$