Separate chaining

- Collision resolution by separate chaining: each slot from the hash table T contains a linked list, with the elements that hash to that slot
- Dictionary operations become operations on the corresponding linked list:
  - $insert(T, x)$ - insert a new node to the beginning of the list $T[h(key[x])]$
  - $search(T, k)$ - search for an element with key k in the list $T[h(k)]$
  - $delete(T, x)$ - delete x from the list $T[h(key[x])]$

Hash table with separate chaining - representation

- A hash table with separate chaining would be represented in the following way (for simplicity, we will keep only the keys in the nodes).

Node:
key: TKey
next: ↑ Node

HashTable:
T: ↑Node[] *//an array of pointers to nodes*
m: Integer
h: TFunction *//the hash function*

Hash table with separate chaining - search

```
function search(ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: function returns True if k is in ht, False otherwise
    position ← ht.h(k)
    currentNode ← ht.T[position]
    while currentNode ≠ NIL and [currentNode].key ≠ k execute
        currentNode ← [currentNode].next
    end-while
    if currentNode ≠ NIL then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Usually search returns the info associated with the key k

Analysis of hashing with chaining

- The average performance depends on how well the hash function h can distribute the keys to be stored among the m slots.
- **Simple Uniform Hashing** assumption: each element is equally likely to hash into any of the m slots, independently of where any other elements have hashed to.
- **load factor** α of the table T with m slots containing n elements
  - is n/m
  - represents the average number of elements stored in a chain
  - in case of separate chaining can be less than, equal to, or greater than 1.

Analysis of hashing with chaining - Insert

- The slot where the element is to be added can be:
  - empty - create a new node and add it to the slot
  - occupied - create a new node and add it to the beginning of the list
- In either case worst-case time complexity is: $\Theta(1)$
- If we have to check whether the element already exists in the table, the complexity of searching is added as well.

Analysis of hashing with chaining - Search I

- There are two cases
  - unsuccessful search
  - successful search
- We assume that
  - the hash value can be computed in constant time ($\Theta(1)$)
  - the time required to search an element with key k depends linearly on the length of the list $T[h(k)]$

Analysis of hashing with chaining - Search II

- **Theorem:** In a hash table in which collisions are resolved by separate chaining, an unsuccessful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.
- **Theorem:** In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.
- Proof idea: $\Theta(1)$ is needed to compute the value of the hash function and α is the average time needed to search one of the m lists

Analysis of hashing with chaining - Search III

- If $n = O(m)$ (the number of hash table slots is proportional to the number of elements in the table, if the number of elements grows, the size of the table will grow as well)
  - $\alpha = n/m = O(m)/m = \Theta(1)$
  - searching takes constant time on average
- Worst-case time complexity is $\Theta(n)$
  - When all the nodes are in a single linked-list and we are searching this list
  - In practice hash tables are pretty fast

Analysis of hashing with chaining - Delete

- If the lists are doubly-linked and we know the address of the node: $\Theta(1)$
- If the lists are singly-linked: proportional to the length of the list

- **All dictionary operations can be supported in $\Theta(1)$ time on average.**
- In theory we can keep any number of elements in a hash table with separate chaining, but the complexity is proportional to α. If α is too large $\Rightarrow$ resize and rehash.

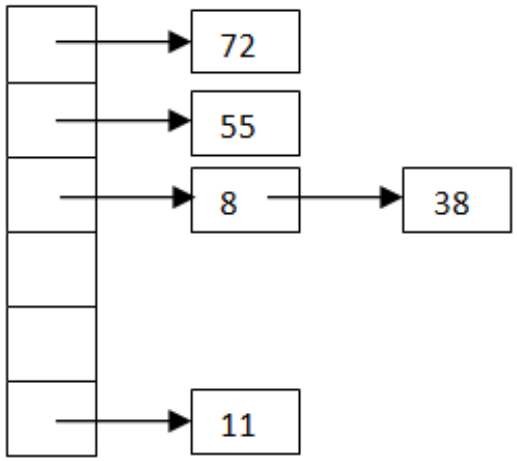
Example

- Assume we have a hash table with m = 6 that uses separate chaining for collision resolution, with the following policy: if the load factor of the table after an insertion is greater than or equal to 0.7, we double the size of the table

- Using the division method, insert the following elements, in the given order, in the hash table: 38, 11, 8, 72, 57, 29, 2.

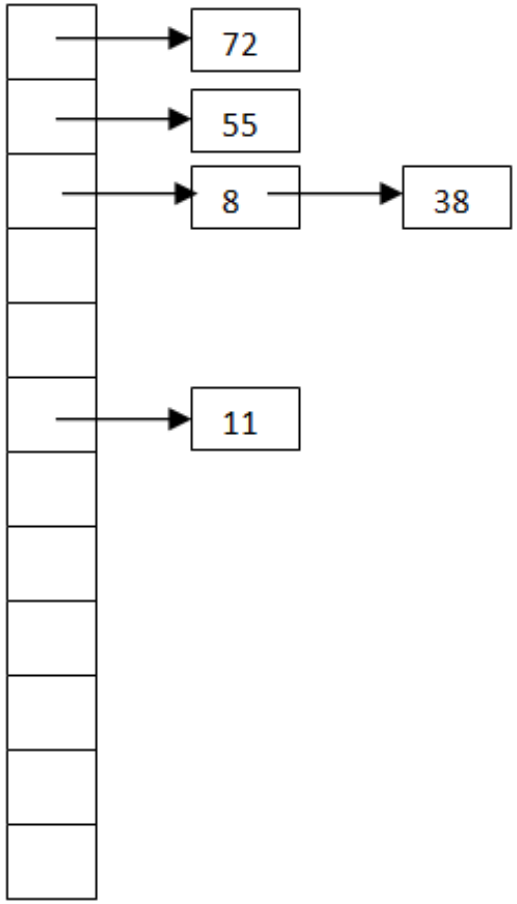
Example

- $h(38) = 2$ (load factor will be $1/6$)
- $h(11) = 5$ (load factor will be $2/6$)
- $h(8) = 2$ (load factor will be $3/6$)
- $h(72) = 0$ (load factor will be $4/6$)
- $h(55) = 1$ (load factor will be $5/6$ - greater than 0.7)
- The table after the first five elements were added:



Example

- Is it OK if after the resize this is our hash table?

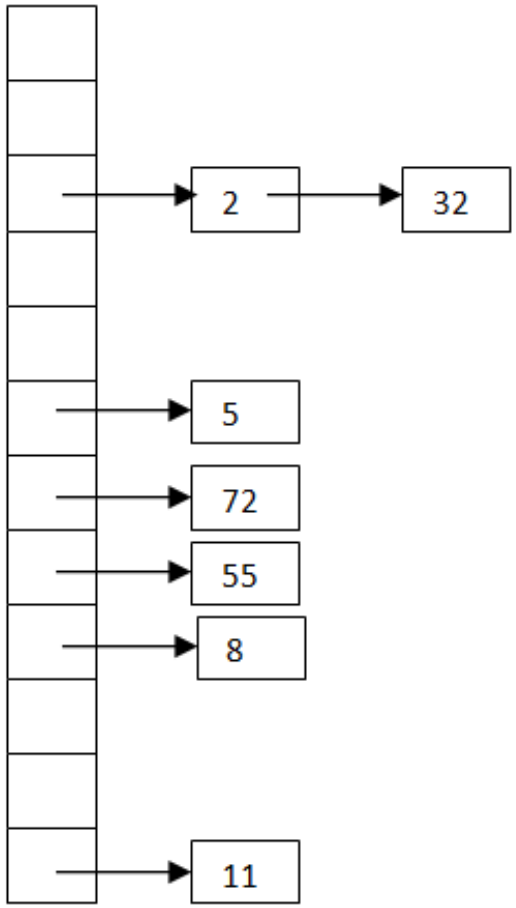


Example

- The result of the hash function (i.e. the position where an element is added) depends on the size of the hash table. If the size of the hash table changes, the value of the hash function changes as well, which means that search and remove operations might not find the element.
- After a resize operation, we have to add all elements again in the hash table, to make sure that they are at the correct position $\rightarrow$ rehash

Example

- After rehash and adding the other two elements:



Iterator

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?
- Since hash tables are used to implement containers where the order of the elements is not important, our iterator can iterate through them in any order.
- For the hash table from the previous example, the easiest order in which the elements can be iterated is: 2, 32, 5, 72, 55, 8, 11

Iterator

- Iterator for a hash table with separate chaining is a combination of an iterator on an array (table) and on a linked list.
- We need a current position to know the position from the table that we are at, but we also need a current node to know the exact node from the linked list from that position.

IteratorHT:
ht: HashTable
currentPos: Integer
currentNode: ↑ Node

Iterator - init

- How can we implement the *init* operation?

```
subalgorithm init(ith, ht) is:
//pre: ith is an IteratorHT, ht is a HashTable
    ith.ht ← ht
    ith.currentPos ← 0
    while ith.currentPos < ht.m and ht.T[ith.currentPos] = NIL execute
        ith.currentPos ← ith.currentPos + 1
    end-while
    if ith.currentPos < ht.m then
        ith.currentNode ← ht.T[ith.currentPos]
    else
        ith.currentNode ← NIL
    end-if
end-subalgorithm
```

- Complexity of the algorithm: $O(m)$

Iterator - other operations

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?
- How can we implement the *valid* operation?

Sorted containers

- How can we define a sorted container on a hash table with separate chaining?
  - Hash tables are in general not very suitable for sorted containers.
  - However, if we have to implement a sorted container on a hash table with separate chaining, we can store the individual lists in a sorted order and for the iterator we can return them in a sorted order.