Data Structures

Week 11 – Stack and queue based on lists; the hash table

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Stack** (week 7) and **queue** (week 8) with arrays: fixed size, a “full” state
- **Linked lists** (weeks 9–10): O(1) at the beginning, with an End pointer at the end too
- **Abstract data type:** the operations matter, not the implementation

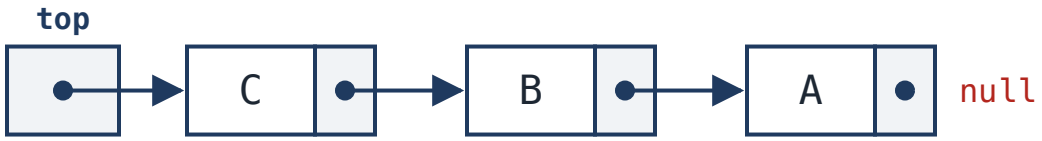
Questions for today

- How can a stack and a queue be implemented with a linked list?
- Can an element be found in **O(1)** time – without sorting and binary search?

Stack and queue with a linked list

Stack with a linked list

Stack as a list: push and pop at the HEAD – both O(1)



`push(x) = insert at head`

`pop() = remove from head`

never 'full'

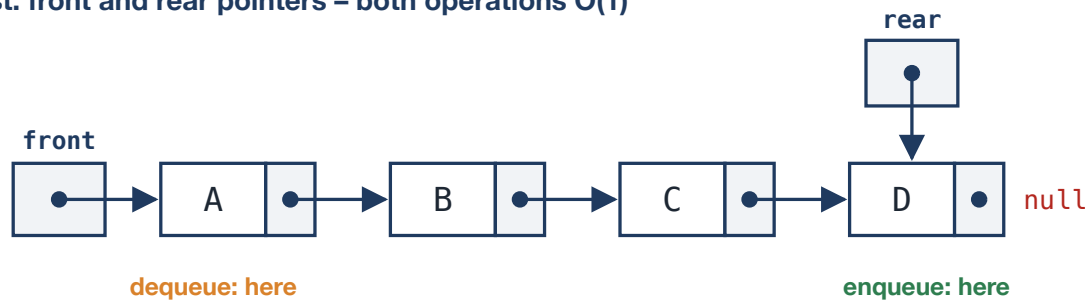
```
Procedure Push(Top, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Top
    Top ← P
  else ERROR
End procedure
```

```
Function Pop(Top)
  If Top = Nil then ERROR
  x ← Elem[Top].Value
  P ← Top
  Top ← Elem[Top].Next
  Free(P)
  Return x
End function
```

This is exactly the “insertion at the beginning” and “deletion from the beginning” of week 9.

Queue with a linked list

Queue as a list: front and rear pointers – both operations O(1)




```
Procedure Enqueue(Front, Rear, Data)
  If Reserve(P) then
    Elem[P].Value ← Data
    Elem[P].Next ← Nil
    If Rear = Nil then Front ← P
    else Elem[Rear].Next ← P
    Rear ← P
  else ERROR
End procedure
```

```
Function Dequeue(Front, Rear)
  If Front = Nil then ERROR
  x ← Elem[Front].Value
  P ← Front
  Front ← Elem[Front].Next
  If Front = Nil then Rear ← Nil
  Free(P)
  Return x
End function
```

When the queue becomes empty, the **Rear** pointer must also be set to Nil!

One interface, two implementations

Java

```java
interface Stack<T> {
    void push(T x);
    T pop();
    boolean isEmpty();
}

class LinkedStack<T> implements Stack<T> {
    private Node<T> top;
    public void push(T x) { top = new Node<>(x, top); }
    public T pop() {
        T x = top.value;
        top = top.next;
        return x;
    }
    public boolean isEmpty() { return top == null; }
}

// the user knows only the interface:
String reverse(String s, Stack<Character> st)
```

Go

```go
type Stack[T any] interface {
    Push(x T)
    Pop() (T, error)
    IsEmpty() bool
}

type LinkedStack[T any] struct{ top *node[T] }

func (s *LinkedStack[T]) Push(x T) {
    s.top = &node[T]{x, s.top}
}

// any implementation can be passed:
reverse("数据结构", &LinkedStack[rune]{})
reverse("数据结构", &SliceStack[rune]{})
```

Array or linked implementation?

	With an array	With a linked list
Push / Pop, Enqueue / Dequeue	O(1)	O(1)
Size	fixed – a “full” state is possible	**unlimited** (while there is memory)
Space per element	only the value	value + **pointer**
Unused space	the empty part of the array	none
Cache-friendly	**yes** (contiguous)	less (scattered)
Allocation per element	none	every push/enqueue allocates

In practice the **dynamic (circular) array** implementation is the most common (Java `ArrayDeque`, Go slice) – fast and memory-efficient.

Hash table

The problem: fast search

Our options so far for finding an element:

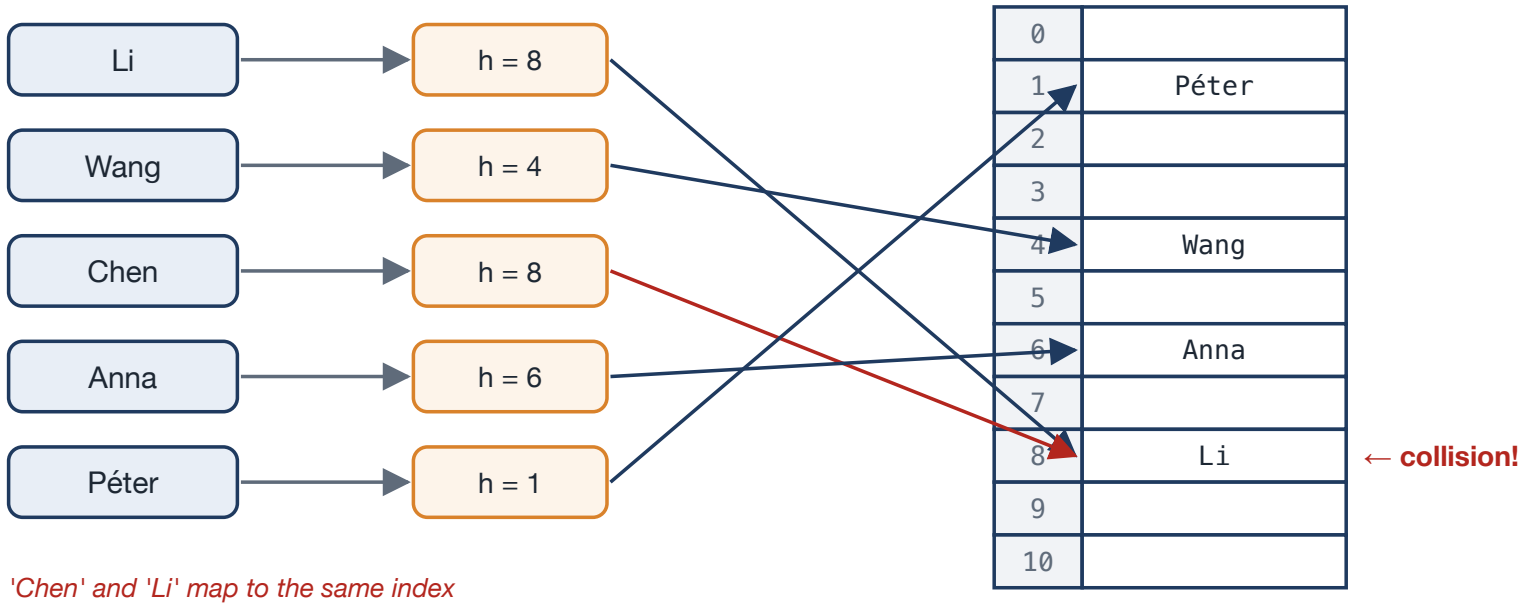
Method	Search	Insertion
unsorted array / list	O(n)	O(1)
sorted array + binary search	O(log n)	O(n)
boolean vector (week 6)	**O(1)**	**O(1)**

The boolean vector is fast, but it only works for a **small, known** universe.

> **Idea:** convert the key (an arbitrarily large number or a string) into a **small index** with a function, and store the element at that index of an array.

Hash function

Hash function: key → index (0 … m – 1), here m = 11



'Chen' and 'Li' map to the same index

A good hash function

Hash function: h: keys → {0, 1, …, m – 1}

Requirements:

- **Fast** to compute – O(length of the key)
- Spreads the keys **evenly** among the buckets
- **Deterministic:** always gives the same result for the same key

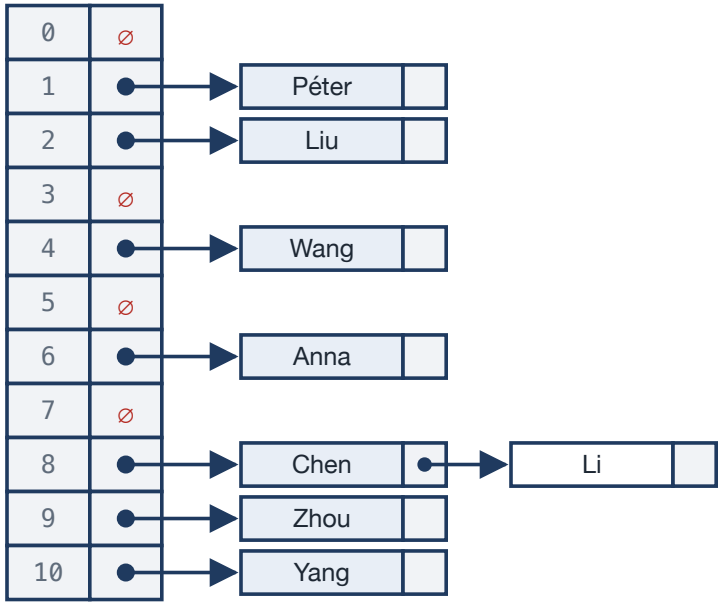
Examples:

- Integer key: **h(k) = k mod m** (m should be a prime)
- String: **polynomial hash**: h = (h · 31 + c) mod m for every character c
  Java: `"Li".hashCode()` = 'L' · 31 + 'i' = 76 · 31 + 105 = **2461**

> **Collision:** two different keys have the same hash value. It is unavoidable (there are more keys than buckets) – it **must be handled**!

Collision handling 1: chaining

Chaining: every bucket is a linked list



insert: at the head of list h(k)

search: only in list h(k)

delete: remove from list h(k)

`load factor: α = n / m = 8 / 11 ≈ 0.73`

average list length = α

Chaining – operations

Pseudocode

```
Procedure Insert(T, key, value)
  i ← h(key)
  P ← Search(T[i], key)        // in the list
  If P ≠ Nil then
    Elem[P].Value ← value      // modification
  else
    InsertFirst(T[i], key, value)
  End if
End procedure

Function Find(T, key)
  Return SearchList(T[h(key)], key)
End function
```

Java

```java
void put(String key, int value) {
    int i = hash(key);
    for (Entry e = buckets[i]; e != null; e = e.next)
        if (e.key.equals(key)) { e.value = value; return; }
    buckets[i] = new Entry(key, value, buckets[i]);
    size++;
    if (size > 0.75 * buckets.length) resize();
}

Integer get(String key) {
    for (Entry e = buckets[hash(key)];
         e != null; e = e.next)
        if (e.key.equals(key)) return e.value;
    return null;
}
```

Load factor and rehashing

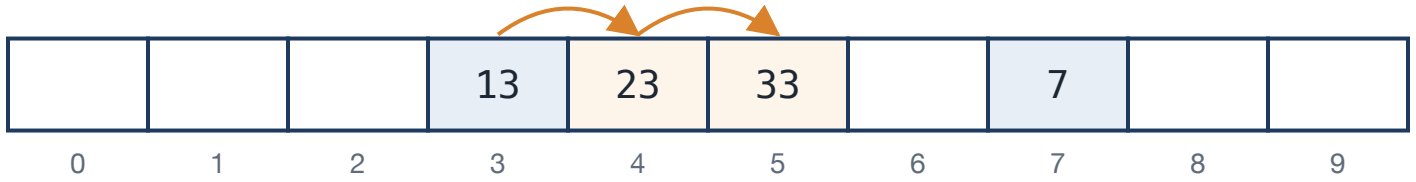
Load factor: α = n / m (number of elements / number of buckets)

- With chaining the average list length is **α** → search is **O(1 + α)** on average
- If α is too large (e.g. > 0.75): **rehashing** – a table twice as large, and **every** element is placed again (the hash depends on **m**!)
- As with the dynamic array: one resize is O(n), but **amortized O(1)**

Example (`HashTable` program): m = 11, 8 elements (α = 0.73) – one collision (Li, Chen)
After the 9th element α = 0.82 > 0.75 → m = 22, α = 0.41 – no collision

Collision handling 2: open addressing

Open addressing, linear probing: h(k) = k mod 10; on collision take the next free slot



`13, 23, 33 all hash to 3 → 3, 4, 5`

search: probe from h(k) until found or an empty slot

- Every element is **in the array itself** – no lists
- On a collision we look for the **next** free slot (linear probing)
- **Deletion:** we cannot simply write “empty” – it would break the search chain → a “deleted” mark is needed

Open addressing: the effect of the load factor

Average number of probes for a successful search (measured values, `LinearProbing` program):

α	0.25	0.50	0.75	0.90	0.99
probes	1.17	1.50	2.48	5.52	**36.16**

- With a low load factor we almost always find it at the **first** try
- As the table fills up, the occupied slots form **clusters** → drastic slowdown
- With open addressing $\alpha < 1$ is mandatory; in practice $\alpha \leq 0.5$–0.75

Efficiency of the hash table

Operation	Average case	Worst case
Search	**O(1)**	O(n)
Insertion	**O(1)** (amortized)	O(n)
Deletion	**O(1)**	O(n)

- Worst case: **every** key goes into the same bucket (a bad hash function, or malicious input)
- **No order:** minimum, maximum, sorted traversal are O(n) / O(n log n)
- Sorted keys need a **search tree** (week 15, Java `TreeMap`)

Built-in hash tables

Java – HashMap, HashSet

```java
Map<String, Integer> freq = new HashMap<>();
for (String w : text.split(" "))
    freq.merge(w, 1, Integer::sum); // freq[w]++

freq.get("fox");                // 2
freq.getOrDefault("cat", 0);    // 0
freq.containsKey("dog");        // true
freq.remove("the");

Set<String> seen = new HashSet<>();
```

The `hashCode()` and `equals()` methods of the key class must be **consistent**!

Go – map

```go
freq := map[string]int{}
for _, w := range strings.Fields(text) {
    freq[w]++          // missing key: 0
}

v, ok := freq["cat"]   // "comma ok"
delete(freq, "the")
len(freq)

seen := map[string]bool{}
```

The iteration order of a map is **random** – on purpose!

Applications: dictionaries, frequencies, caches, database indexes, filtering duplicates, symbol tables of compilers

Summary

- **Stack with a list:** push/pop at the beginning of the list · **queue with a list:** Front + Rear pointers – all O(1), no “full” state
- The same **interface** with several implementations
- **Hash table:** an index from the key with a hash function → search, insertion, deletion **O(1)** on average
- **Collision handling:** chaining (a list per bucket) or open addressing (next free slot)
- **Load factor** α = n/m; too large α → rehashing
- Java `HashMap` / `HashSet` · Go `map`

Next week: recursion.

Review questions

1. Why does the linked stack work at the **beginning** of the list and not at the end?
2. What happens in the linked queue if we remove the last element and forget to set the Rear pointer to Nil?
3. h(k) = k mod 7, with chaining. Insert: 10, 3, 17, 24, 5! Draw the table!
4. The same keys with linear probing, m = 7 – where does 24 go?
5. True or false: the search time of a hash table is O(1) even in the worst case.
6. Why can the smallest key not be found efficiently in a hash table?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Linked stack and queue	`LinkedStackQueue.java`	`w11/linkedstackqueue`	interface, two implementations
Hash table	`HashTable.java`	`w11/hashtable`	chaining, collision, rehashing
Linear probing	`LinearProbing.java`	`w11/linearprobing`	open addressing, effect of the load factor
Word frequency	`WordCount.java`	`w11/wordcount`	using `HashMap` / `map`

✎ Exercises

1. **Two sum:** are there two elements in the array whose sum is a given value? O(n) with a hash table instead of $O(n^2)$!
2. **First unique character:** which is the first character that occurs only once? (`"swiss" → 'w'`)
3. **Anagram groups:** group the words so that anagrams of each other are in the same group!
4. **Duplicates:** is there a repeated element in the array? (O(n) on average)
5. **Brackets:** the task of week 7, with the bracket pairs stored in a map.
6. **Phone book:** name → phone number, search by name.

Solutions: `Exercises11.java`, `w11/exercises`

✎ Homework

1. Extend our own `HashTable` class with a `keys()` method that returns every key! How many steps does it take?
2. Implement **deletion** in the linear probing table with a “deleted” mark! How do search and insertion change?
3. Measure the number of collisions of your own hash table for 10 000 random words with two hash functions: (a) the sum of the character codes, (b) polynomial hash! What do you observe?
4. Implement an **LRU cache** with a hash table and a doubly linked list (see the homework of week 10), where both `get` and `put` are O(1)!