Collections

Kamal kumar mukiri, Apt Computing Labs

COLLECTIONS

- Vectors
- Strings
- HashMap
- BTreeMap
- HashSet
- BTreeSet
- VecDeque
- BinaryHeap

Just a glance

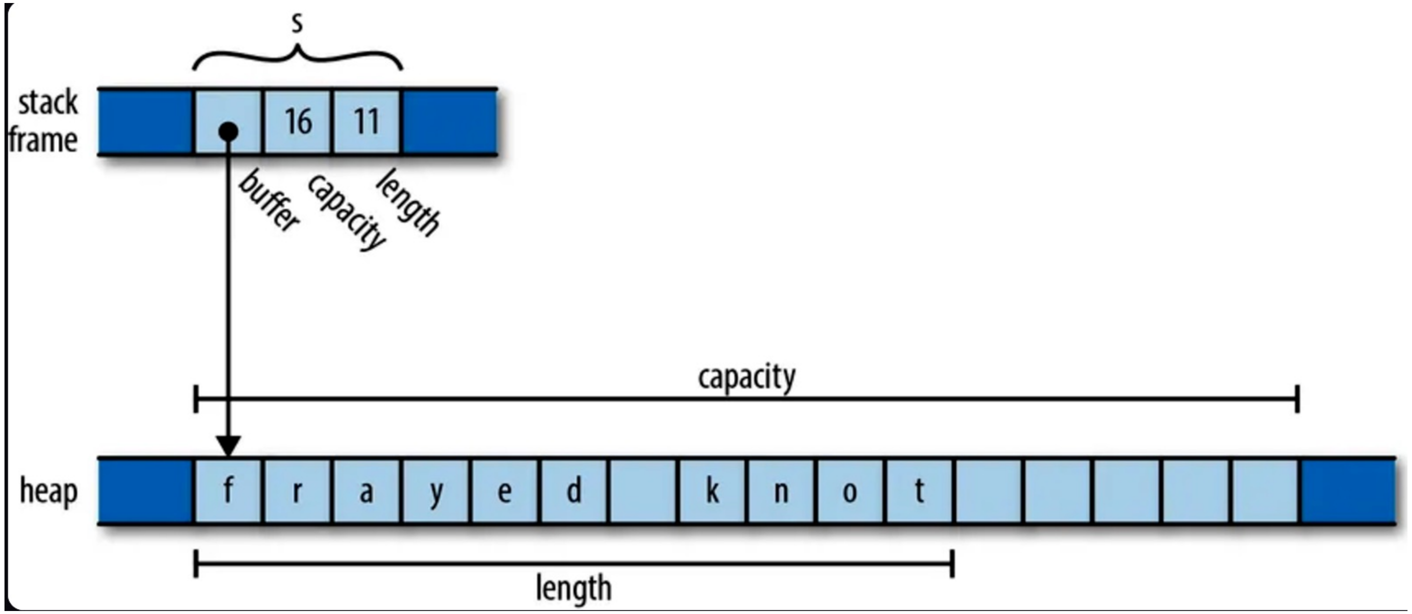
Collection	Best Use Case	Features
Vec	Dynamic arrays	Fast random access
String	Text manipulation	UTF-8 encoded
HashMap	Key-value pairs, unordered	Fast lookups, unordered
BTreeMap	Key-value pairs, sorted	Maintains order of keys
HashSet	Unique items, unordered	Fast membership checks
BTreeSet	Unique items, sorted	Maintains order of items
VecDeque	Double-ended queue	Efficient at both ends
BinaryHeap	Priority queue	Max heap-based operations

Collection	When to choose	Typical complexities (Big-O)
`Vec<T>`	Growable contiguous list; fast random access; default dynamic sequence.	index `O(1)`; push_back `amortized O(1)`; pop_back `O(1)`; insert/remove in middle `O(n)`; search `O(n)`; iterate `O(n)`
`VecDeque<T>`	Queue/deque with O(1) push/pop at both ends; ring buffer.	index `O(1)`; push_front/back `O(1)`; pop_front/back `O(1)`; insert/remove in middle `O(n)`; iterate `O(n)`
`LinkedList<T>`	Rare; frequent mid-list insert/remove when you already have a cursor/node.	push_front/back `O(1)`; pop_front/back `O(1)`; insert/remove **at cursor** `O(1)`; search/index by position `O(n)`; iterate `O(n)`
`BinaryHeap<T>`	Priority queue; repeatedly get/remove max (or min via Reverse).	push `O(log n)`; pop_max `O(log n)`; peek `O(1)`; build from vec `O(n)`; contains/search `O(n)`; iterate (unsorted) `O(n)`

Collection	When to choose	Typical complexities (Big-O)
`HashMap<K,V>`	Unordered key→value; fastest average-case lookups/inserts; no sorted iteration.	**avg:** insert/lookup/remove `O(1)`; **worst:** `O(n)`; iterate `O(n)`
`BTreeMap<K,V>`	Sorted key→value; ordered iteration, range queries.	lookup/insert/remove `O(log n)`; iterate sorted `O(n)`; range `[a,b]` `O(log n + k)`
`HashSet<T>`	Unordered unique items; fast membership tests.	**avg:** insert/contains/remove `O(1)`; **worst:** `O(n)`; iterate `O(n)`
`BTreeSet<T>`	Sorted unique items; need ordering or range queries.	contains/insert/remove `O(log n)`; iterate sorted `O(n)`; range `[a,b]` `O(log n + k)`
`String`	Owned, growable UTF-8 text; build/modify strings.	push_back `amortized O(1)`; insert/remove at pos `O(n)`; index by byte `O(1)` (*only at char boundaries*); substring slice `O(1)`; search `O(n)`; iterate chars `O(n)`

Strings: Memory structure

-



Strings: Introduction

●

•A **growable, UTF-8 encoded text string**.

•Allows efficient string manipulation.

Key Methods

.push_str("text") – Appends text.

.push('c') – Appends a character.

.len() – Returns the number of bytes (not characters).

.replace("old", "new") – Replaces substrings.

Strings: Sample code

-

```
fn main() {
    let mut s1: String = String::from("Sample text");
    s1.push_str("... Added text");
    println!("Final string = {}", s1);
}
```

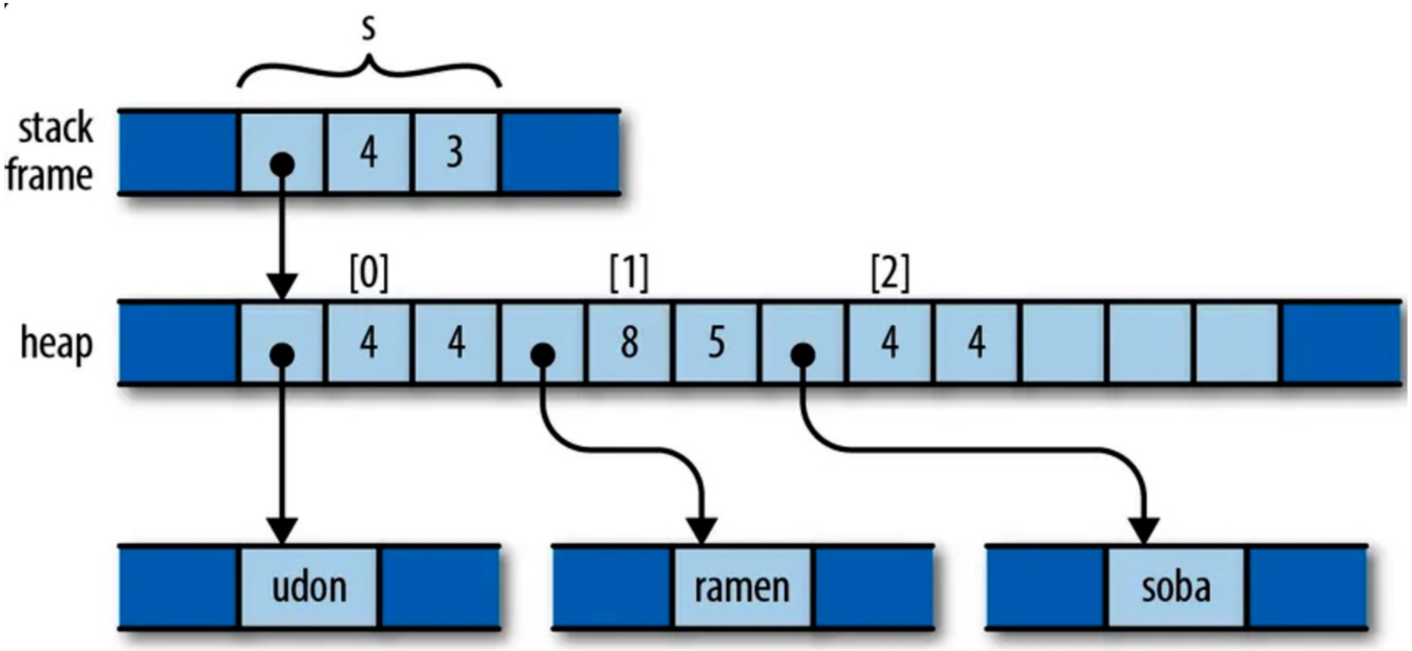
Strings: Frequently used methods

-

Method	Description
`len()`	Length in bytes
`is_empty()`	True if length is zero
`as_bytes()`	View as byte slice
`chars()`	Iterate Unicode chars
`contains(pat)`	Substring/pattern exists
`starts_with(pat)`	Prefix match
`ends_with(pat)`	Suffix match
`find(pat)`	First index of pattern
`split_whitespace()`	Split by Unicode whitespace
`split(pat)`	Split by pattern
`lines()`	Iterate over lines
`trim()`	Strip both ends

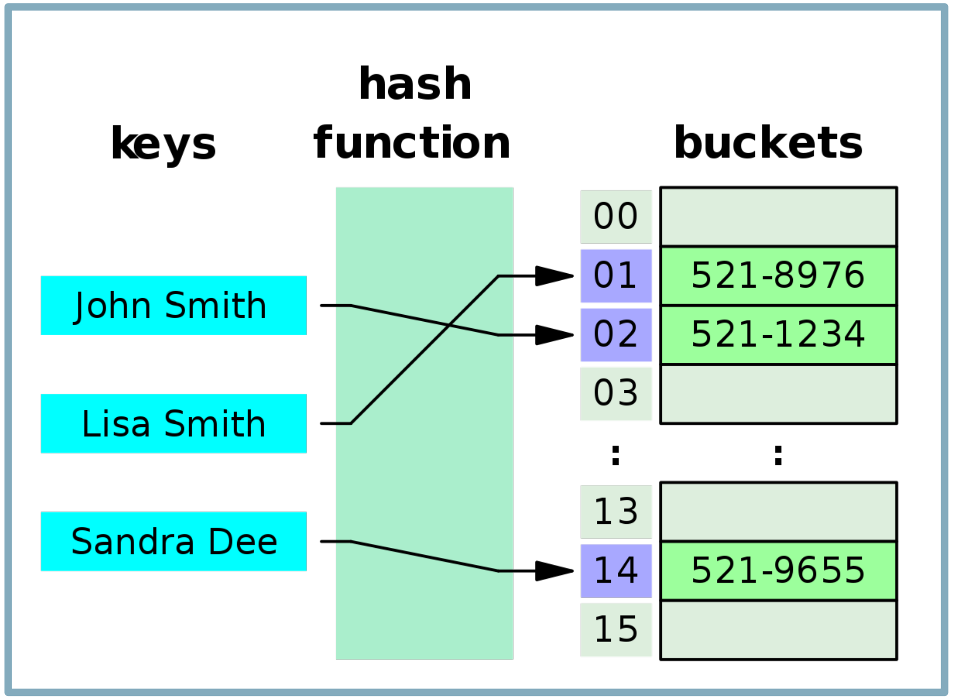
https://doc.rust-lang.org/std/string/struct.String.html

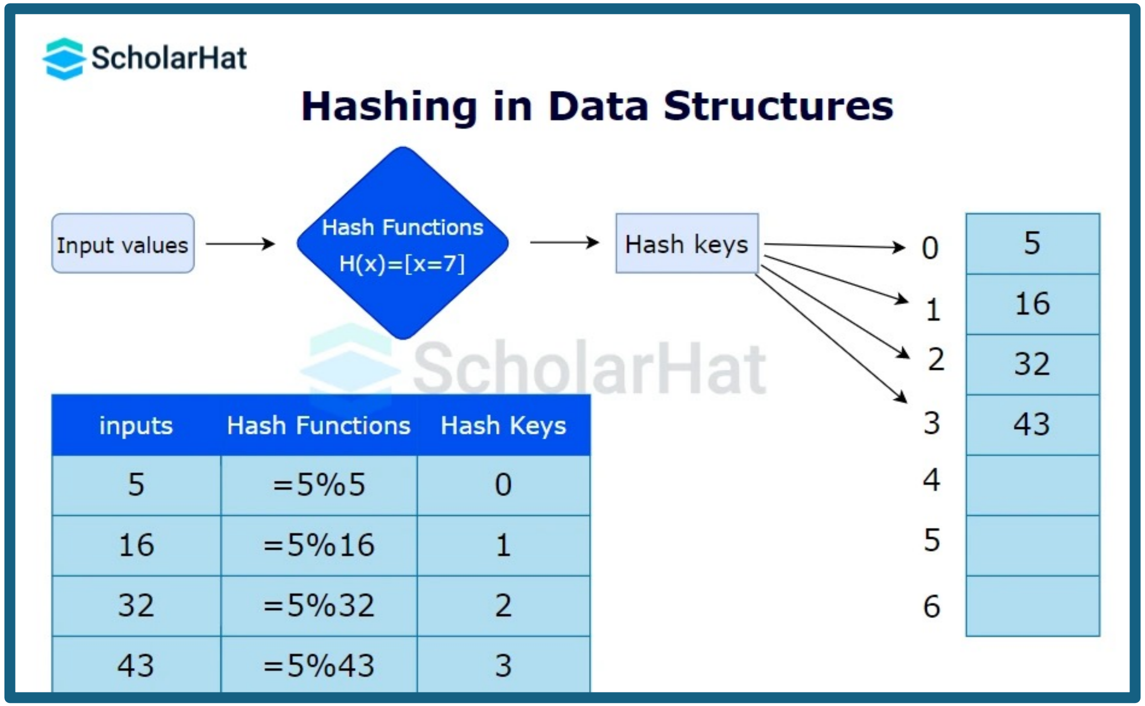
Vectors: Memory structure



HashMap: Memory structure

-





inputs	Hash Functions	Hash Keys
5	=5%5	0
16	=5%16	1
32	=5%32	2
43	=5%43	3

HashMap: Introduction

- A **key-value store**.
- Allows fast lookups by hashing keys.
- Keys and values can be of any type that implements Eq and Hash.

Key Methods

`.insert(key, value)` – Adds a key-value pair.
`.get(&key)` – Returns a reference to the value.
`.remove(&key)` – Removes a key-value pair.
`.contains_key(&key)` – Checks if a key exists.

HashMap: Sample code

```rust
use std::collections::HashMap;

fn main() {
    let mut scores: HashMap<&str, i32> = HashMap::new();
    scores.insert("Alice", 50);
    scores.insert("Bob", 40);

    if let Some(&score) = scores.get("Alice") {
        println!("Alice's score: {}", score);
    }

    scores.remove("Bob");
    println!("{:?}", scores); // {"Alice": 50}
}
```

1. std::collections::HashMap
2. Creating an empty Hash
3. Inserting a “key-value” pair
4. Removing a pair

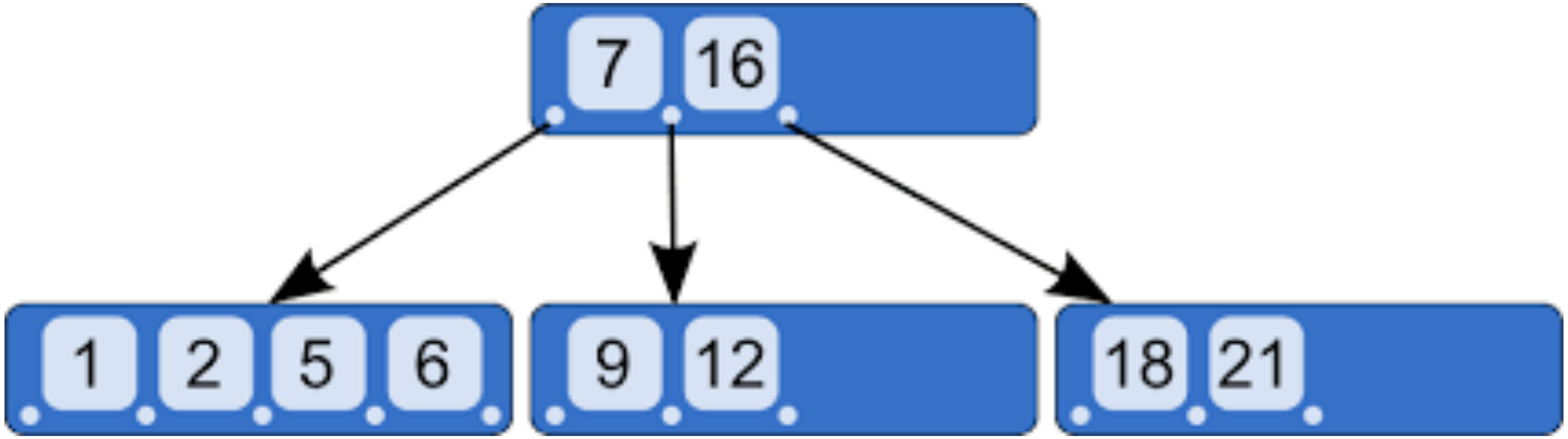
HashMap: Sample code

-

```
use std::collections::HashMap;

fn main() {
    let mut map1 = HashMap::new();

    map1.insert("Bats Men", vec!["Kohli","Surya"]);
    map1.insert("Bowlers", vec!["Bhumra", "Kapil"]);
    map1.insert("team", vec!["Bhumra", "Kapil"]);
    map1.insert("crew", vec!["Bhumra", "Kapil"]);
    map1.insert("helpers", vec!["Bhumra", "Kapil"]);

    for (key, value) in map1 {
        println!("Key ={}, Value = {:?}", key, value);
    }
}
```

BTreeMap: Memory structure

-



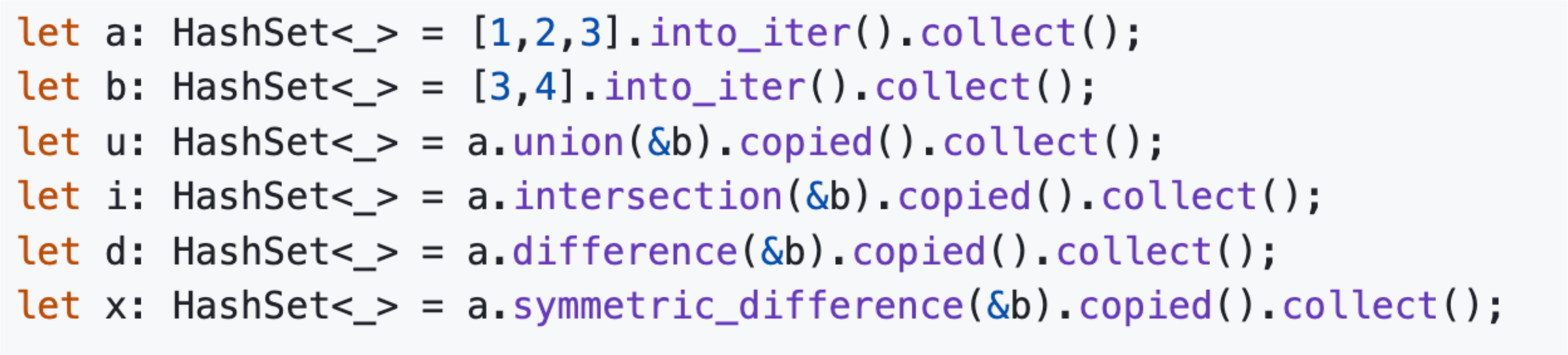
BTreeMap: Introduction

- Similar to HashMap, but maintains **sorted order** of keys.
- Useful when order is important.

BTreeMap: Sample code

```
use std::collections::BTreeMap;


fn main() {
    let mut t1 = BTreeMap::new();
    t1.insert(”Kapil", vec![1,2,3]);
    t1.insert("David", vec![1,2,3]);
    t1.insert("Azad", vec![1,2,3]);
    t1.insert("Sri", vec![1,2,3]);




    for (key, value) in t1 {
        println!("Key: {}, Value: {:?}", key, value);
    }
}
```

HashSet: Sample code



```rust
let a: HashSet<_> = [1,2,3].into_iter().collect();
let b: HashSet<_> = [3,4].into_iter().collect();
let u: HashSet<_> = a.union(&b).copied().collect();
let i: HashSet<_> = a.intersection(&b).copied().collect();
let d: HashSet<_> = a.difference(&b).copied().collect();
let x: HashSet<_> = a.symmetric_difference(&b).copied().collect();
```