

Data Storage / Collections

All data stored on the NEAR blockchain is done in key / value pairs. There are several collection methods in the SDKs we've created that will help you store your data on chain.

- [near-sdk-rs](#)
- for [Rust](#)
- smart contracts
- [near-sdk-js](#)
- for [JavaScript](#)
- smart contracts

For information on storage costs, please see [storage staking](#) .

Rust Collection Types

[near-sdk-rs module documentation](#)

Type Iterable Clear All Values Preserves Insertion Order Range Selection [Vector](#) ✓ ✓ ✓ ✓ [LookupSet](#) [UnorderedSet](#) ✓ ✓

✓ [LookupMap](#) [UnorderedMap](#) ✓ ✓

✓ [TreeMap](#) ✓ ✓

Big-O Notation

The [Big-O notation](#) values in the chart below describe the [time complexity](#) of the various collection methods found in [near-sdk-rs](#) . These method complexities correlate with [gas](#) consumption on NEAR, helping you decide which collection to utilize in your project. There are three types found in our collection methods: * $O(1)$ - [constant](#) * $O(n)$ - [linear](#) * $O(\log n)$ - [logarithmic](#)

Type	Access	Insert	Delete	Search	Traverse	Clear	Vector	$O(1)$	$O(1)$	$O(1)^*$	$O(n)$	$O(n)$	$O(n)$	LookupSet	$O(1)$	$O(1)$	$O(1)$	$O(1)$	N/A
N/A	UnorderedSet	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	LookupMap	$O(1)$	$O(1)$	$O(1)$	$O(1)$	N/A	N/A	UnorderedMap	$O(1)$	$O(1)$	$O(1)$	
$O(1)$	$O(n)$	$O(n)$	TreeMap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	*	- to insert at the end of the vector using <code>push_back</code> (or <code>push_front</code> for deque)								

** - to delete from the end of the vector using `pop` (or `pop_front` for deque), or delete using `swap_remove` which swaps the element with the last element of the vector and then removes it.

Gas Consumption Examples

The examples below show differences in gas burnt storing and retrieving key/value pairs using the above methods. Please note that the gas cost of spinning up the runtime environment on chain has been deducted to show just data read/writes.

You can reproduce this and test out your own data set by visiting [collection-examples-rs](#) .

Vector

Implements [a vector](#) / persistent array.

- can iterate using index
- Uses the following map: index -> element. [SDK source](#)]

[\[Implementation\]](#)]

LookupSet

Implements a persistent set without iterators.

- can not iterate over keys
- more efficient in reads / writes [SDK source](#)]

[\[Implementation\]](#)]

UnorderedSet

Implements a persistent set with iterators for keys, values, and entries. [SDK source](#)]

[\[Implementation Docs\]](#)

LookupMap

Implements a persistent map.

- can not iterate over keys
- does not preserve order when removing and adding values
- efficient in number of reads and writes
- To add data:

```
pub
fn
add_lookup_map ( & mut
self , key :
String , value :
String )
{ self . lookup_map . insert ( & key ,
& value ) ; } * To get data:
pub
fn
get_lookup_map ( & self , key :
String )
->
String
{ match
self . lookup_map . get ( & key )
{ Some ( value )
=>
{ let log_message =
format! ( "Value from LookupMap is {:?}" , value . clone ( ) ) ; env :: log ( log_message . as_bytes ( ) ) ; value } , None
=>
"not found" . to_string ( ) } } SDK source ]
```

[\[Implementation\]](#)

UnorderedMap

Implements an unordered map.

- iterable
- does not preserve order when removing and adding values
- is able to clear all values
- To add data:

```

pub
fn
add_unordered_map ( & mut
self , key :
String , value :
String )
{ self . unordered_map . insert ( & key ,
& value ) ; } * To get data:
pub
fn
get_unordered_map ( & self , key :
String )
->
String
{ match
self . unordered_map . get ( & key )
{ Some ( value )
=>
{ let log_message =
format! ( "Value from UnorderedMap is {:?}" , value . clone ( ) ) ; env :: log ( log_message . as_bytes ( ) ) ; value } , // None
=> "Didn't find that key.".to_string() None
=>
"not found" . to_string ( ) } } SDK source ]
\[Implementation\] ]

```

TreeMap

Implements a Tree Map based on [AVL-tree](#) .

- iterable
- preserves order
- able to clear all values
- self balancing
- To add data:

```

pub
fn
add_tree_map ( & mut
self , key :
String , value :
String )
{ self . tree_map . insert ( & key ,

```

& value) ; } * To get data:

pub

fn

get_tree_map (& self , key :

String)

->

String

{ match

self . tree_map . get (& key)

{ Some (value)

=>

{ let log_message =

format! ("Value from TreeMap is {:?}" , value . clone ()) ; env :: log (log_message . as_bytes ()) ; // Since we found it, return it (note implicit return) value } , // did not find the entry // note: curly brackets after arrow are optional in simple cases, like other languages None

=>

"not found" . to_string () } } [\\$SDK source](#)]

[\[Implementation\]](#)]

Storage Constraints on NEAR

For storing data on-chain it's important to keep in mind the following:

- Can add up in storage staking costs
- There is a 4mb limit on how much you can upload at once

Let's say for example, someone wants to put an NFT purely on-chain (rather than IPFS or some other decentralized storage solution) you'll have almost an unlimited amount of storage but will have to pay 1 NEAR per 100kb of storage used (see Storage Staking).

Users will be limited to 4MB per contract call upload due to MAX_GAS constraints. The maximum amount of gas one can attach to a given functionCall is 300TGas. [Edit this page](#) Last updated on Jan 26, 2023 by Dorian Crutcher Was this page helpful? Yes No

[Previous](#) [Avoiding Token Loss](#) [Next](#) [Storage Staking](#)