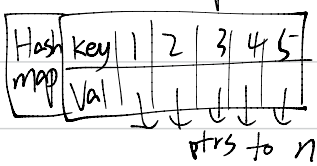
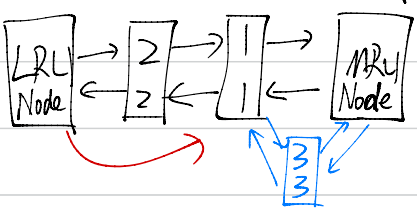
Intuition:

① use hashmap to save key-value pairs



② use double-linked list to keep track of LRU and MRU (most recent used)



if need to remove node (due to capacity restraints), then link next pointer of the LRU Node to the node→next

Similarly, always add node from the right by changing pointers

③ with C++, we can use STL list functions to easily achieve this

- key functions:

std::list<T,Allocator>::pop_back

```
void pop_back();
```

Removes the last element of the container.

Calling pop_back on an empty container results in undefined behavior.

References and iterators to the erased element are invalidated.

Parameters

(none)

Return value

(none)

Complexity

Constant.

Exceptions

Throws nothing.

std::list<T,Allocator>::emplace_front

```
template< class... Args >                    (since C++11)
void emplace_front( Args&&... args );        (until C++17)

template< class... Args >                    (since C++17)
reference emplace_front( Args&&... args );
```

Inserts a new element to the beginning of the container. The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at the location provided by the container. The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`.

No iterators or references are invalidated.

Parameters

args - arguments to forward to the constructor of the element

Type requirements

- `T (the container's element type)` must meet the requirements of *EmplaceConstructible*.

Return value

(none) (until C++17)
A reference to the inserted element. (since C++17)

pop back and emplace_front ensures we always remove from the back and add from beginning then, we just need to ensure, every time we update or get value of a node, it gets moved to the beginning

std::list<T,Allocator>::splice

`void splice( const_iterator pos, list& other );`	(1)	
`void splice( const_iterator pos, list&& other );`	(1)	(since C++11)
`void splice( const_iterator pos, list& other, const_iterator it );`	(2)	
`void splice( const_iterator pos, list&& other, const_iterator it );`	(2)	(since C++11)
`void splice( const_iterator pos, list& other, const_iterator first, const_iterator last);`	(3)	
`void splice( const_iterator pos, list&& other, const_iterator first, const_iterator last );`	(3)	(since C++11)

Transfers elements from one list to another.

No elements are copied or moved, only the internal pointers of the list nodes are re-pointed. The behavior is undefined if: `get_allocator() != other.get_allocator()`. No iterators or references become invalidated, the iterators to moved elements remain valid, but now refer into `*this`, not into `other`.

1) Transfers all elements from `other` into `*this`. The elements are inserted before the element pointed to by `pos`. The container `other` becomes empty after the operation. The behavior is undefined if `other` refers to the same object as `*this`.

2) Transfers the element pointed to by `it` from `other` into `*this`. The element is inserted before the element pointed to by `pos`.

3) Transfers the elements in the range `[first, last)` from `other` into `*this`. The elements are inserted before the element pointed to by `pos`. The behavior is undefined if `pos` is an iterator in the range `[first,last)`.

Parameters

pos	-	element before which the content will be inserted
other	-	another container to transfer the content from
it	-	the element to transfer from `other` to `*this`
first, last	-	the range of elements to transfer from `other` to `*this`

Return value

(none)

Exceptions

Throws nothing.

Complexity

1-2) Constant.

3) Constant if `other` refers to the same object as `*this`, otherwise linear in `std::distance(first, last)`.