

Think about:

Design a data structure with the following properties :

- For search, insert and remove , it has the same complexity as in the case of a hash table
- The iteration order is the insertion order. Operations of the iterator are in $\Theta(1)$

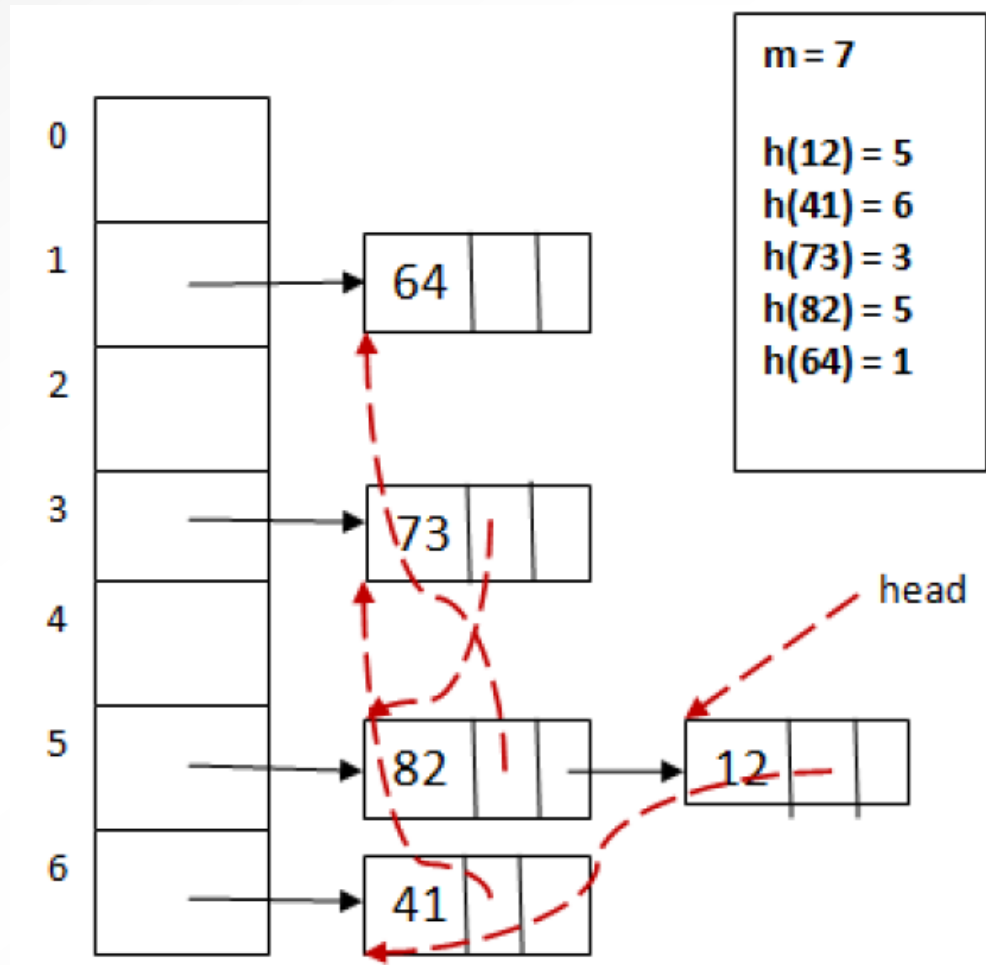
Give:

- Representation
- Draw the DS containing elements 12, 41, 73, 82, 64 , added in this order

Is the next example drawing correct?

Assume: $m=7$, division method for hash function

(For the things that are not specified, we choose a “good enough” value and/or method)



Representation:

What are the differences from the example before ?

Node:

info: TKey

nextH: \uparrow Node

nextL: \uparrow Node

prevL: \uparrow Node

//pointer to next node from the collision

//pointer to next node from the insertion-order list

//pointer to prev node from the insertion-order list

LinkedHT:

T: (\uparrow Node)[]

m: Integer

h: TFunction

head: \uparrow Node

tail: \uparrow Node

Think about:

How can we represent the Iterator ?

Linked Hash Table

Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.
 - It has a predictable iteration order \leq linked list
- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient?
 - Think about the operations that we usually have for a hash table

Linked Hash Table - insert

subalgorithm insert(lht, k) is:

allocate(newNode)

[newNode].info \leftarrow k

@set all pointers of newNode to NIL

pos \leftarrow lht.h(k)

if lht.T[pos] = NIL then

lht.T[pos] \leftarrow newNode

else

[newNode].nextH \leftarrow lht.T[pos]

lht.T[pos] \leftarrow newNode

end-if

if lht.head = NIL then

lht.head \leftarrow newNode

lht.tail \leftarrow newNode

else

[newNode].prevL \leftarrow lht.tail

[lht.tail].nextL \leftarrow newNode

lht.tail \leftarrow newNode

end-if

end-subalgorithm

- insert newNode into the hash table

- insert newNode to the end of the insertion-order list

Linked Hash Table - remove

subalgorithm remove(lht, k) is:

pos \leftarrow lht.h(k)

current \leftarrow lht.T[pos]

nodeToBeRemoved \leftarrow NIL

if current \neq NIL and [current].info = k then

nodeToBeRemoved \leftarrow current

 lht.T[pos] \leftarrow [current].nextH

else

 prevNode \leftarrow NIL

 while current \neq NIL and [current].info \neq k execute

 prevNode \leftarrow current

 current \leftarrow [current].nextH

 endwhile

 if current \neq NIL then

nodeToBeRemoved \leftarrow current

 [prevNode].nextH \leftarrow [current].nextH

 else

 @k is not in lht

 end-if

end-if

search for k in the
collision list and
remove it if found

```

if nodeToBeRemoved ≠ NIL then
    if nodeToBeRemoved = lht.head then
        if nodeToBeRemoved = lht.tail then
            lht.head ← NIL
            lht.tail ← NIL
        else
            lht.head ← [lht.head].nextL
            [lht.head].prev ← NIL
        end-if
    else
        if nodeToBeRemoved = lht.tail then
            lht.tail ← [lht.tail].prev
            [lht.tail].next ← NIL
        else
            [[nodeToBeRemoved].next].prev ← [nodeToBeRemoved].prev
            [[nodeToBeRemoved].prev].next ← [nodeToBeRemoved].next
        end-if
    end-if
    free (nodeToBeRemoved)
end-if
end-subalgorithm

```

- if *k* was found, then *nodeToBeRemoved* is the address of the node containing it
- we need to remove it from the insertion-order list

Other DS. Linked Structures

- Multiple Links per node

Node:

info: TElem

links: List< \uparrow Node>

Terminology:

general linked lists

multi-linked lists

- Multiple Values per node

ULNode:

next: \uparrow ULNode

elemCount: Integer

elemData: TElem [MAX]

Terminology:

unrolled linked list

Other DS. Linked structures

Think about:

1. We want to organize a collection of elements in two different ways.

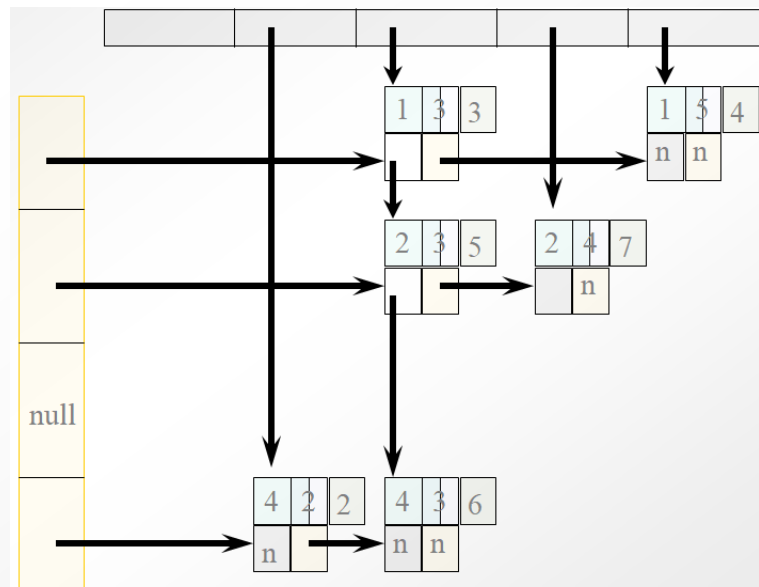
Suppose our data include the name of a person and his/her age.

e.g.: (FRED,19) (MARY,16) (JACK,21) (JOHN,18)

We want to keep a SLL having these elements sorted alphabetically and also sorted by age. How could we do this?

2. Remember sparse matrices.

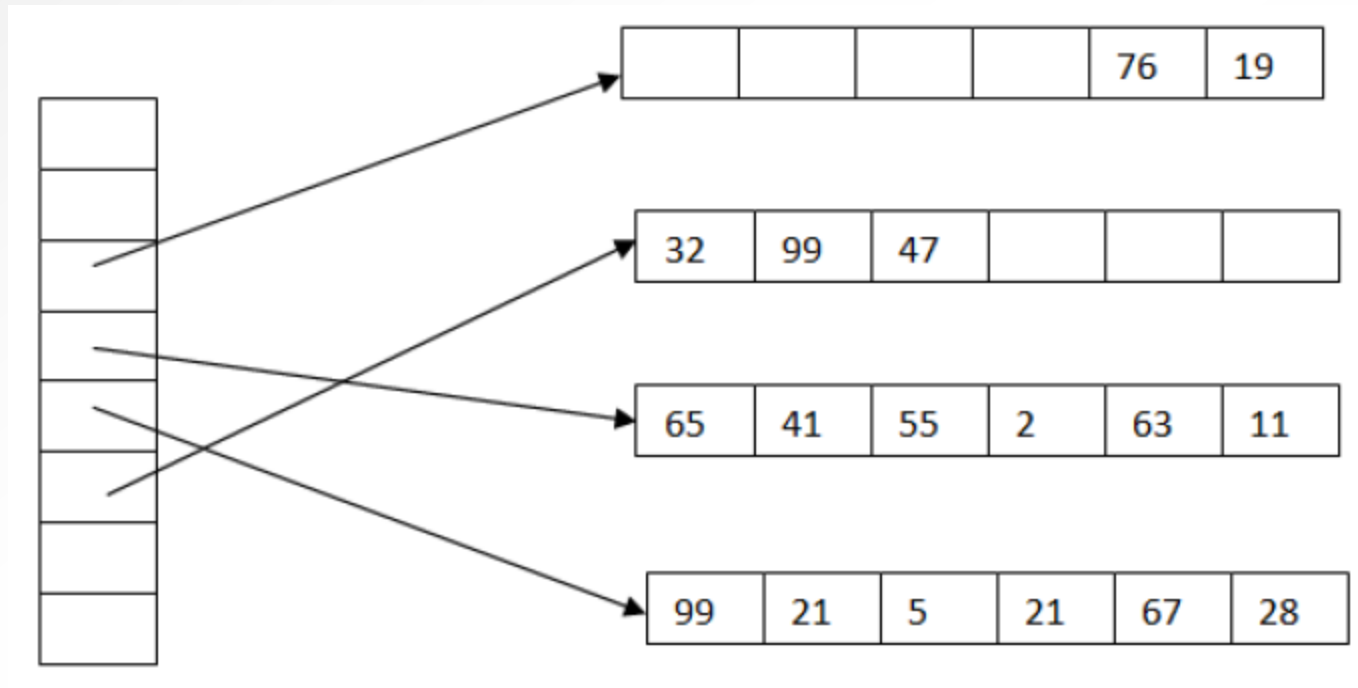
How can we define
a representation corresponding
to the next example?



Think about:

Assume a container with the next elements

76, 19, 65, 41, 55, 2, 63, 11, 99, 21, 5, 21, 67, 28, 32, 99, 47
with a structure as in the next drawing:



1. Describe a possible representation for such a structure.
 2. What could be the time complexity for accessing the individual elements based on index?
- What can you say about insertion and deletion of elements at the beginning and at the end?

Deque C++

Source:

<https://www.cplusplus.com/reference/deque/deque/>

“Specific libraries may implement deques in different ways, generally as some form of dynamic array.”

“Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways:

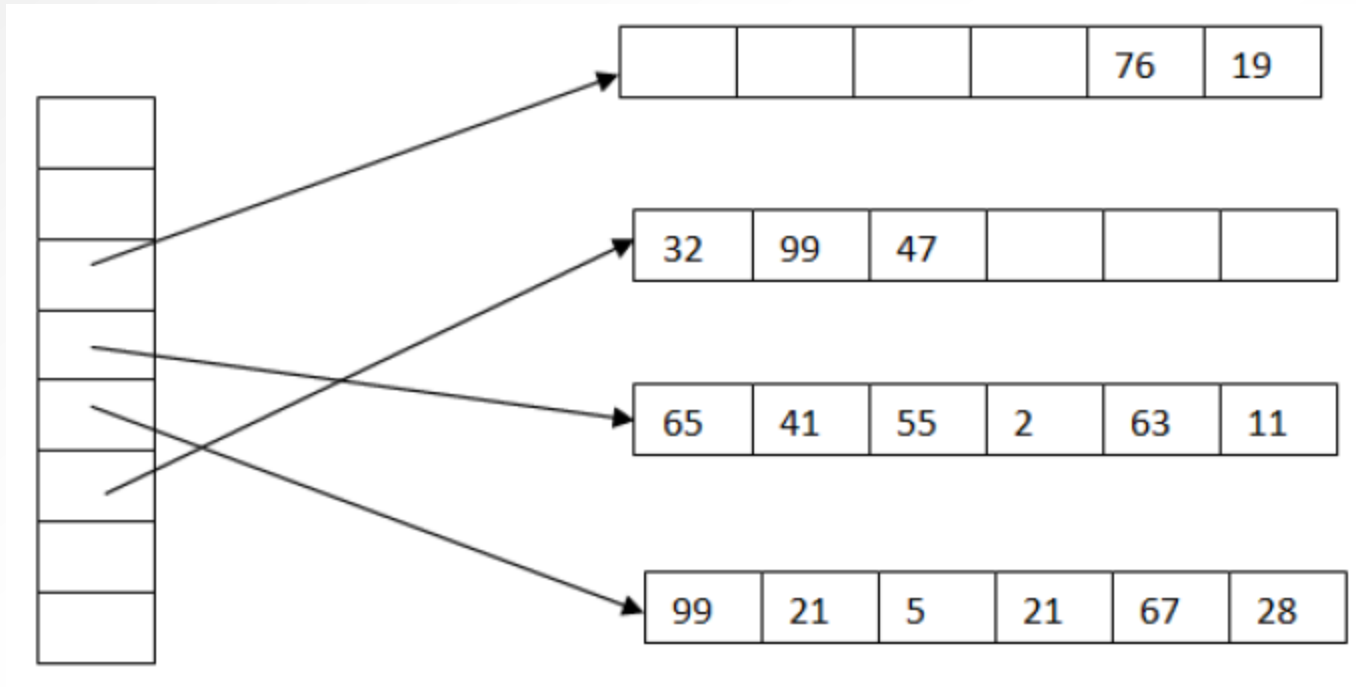
While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little more complex internally than vectors, but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.”

“For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than lists and forward lists.”

A representation for a deque is to use a dynamic array of fixed size arrays

- Place the elements in fixed size arrays (blocks).
- Keep a dynamic array with the addresses of these blocks.
- Every block is full, except for the first and last ones.
- The first block is filled from right to left.
- The last block is filled from left to right.
- If the first or last block is full, a new one is created and its address is put in the dynamic array.
- If the dynamic array is full, a larger one is allocated, and the addresses of the blocks are copied (but elements are not moved)

A representation for a deque can be:
use a dynamic array of fixed size arrays



Elements of the deque:

76, 19, 65, ..., 11, 99, ..., 28, 32, 99, 47

A representation for a deque is to use a dynamic array of fixed size arrays

Information (fields) we need to represent a deque using a dynamic array of blocks:

- Block size
- The dynamic array with the addresses of the blocks
- Capacity of the dynamic array
- First occupied position in the dynamic array
- First occupied position in the first block
- Last occupied position in the dynamic array
- Last occupied position in the last block

.

Think about:

- How can we implement a Stack using two Queues?
What will be the complexity of the operations?
- How can we implement a Queue using two Stacks?
What will be the complexity of the operation?
- How can we implement two Stacks using only one array?

The stack operations should throw an exception only if the total number of elements in the two Stacks is equal to the size of the array.

- How can we implement a stack using a Priority Queue?
- How can we implement a queue using a Priority Queue?