Data Structures

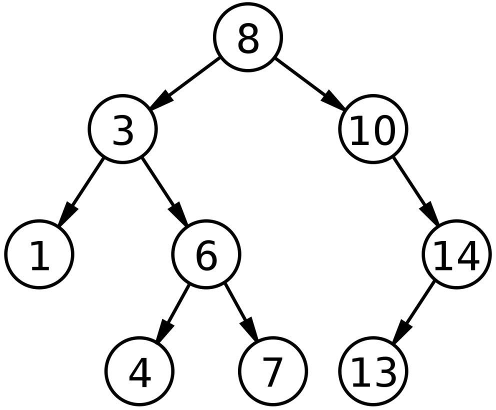
Data Structures

“A way to store and arrange data, in a way that suits an algorithm.”

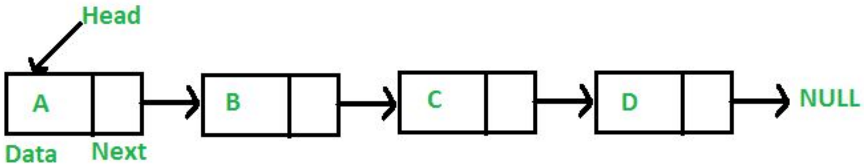
Has a concrete implementation: we can calculate **complexity** of operations

Examples:

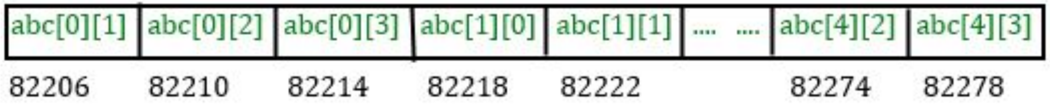
- Arrays
- Linked Lists
- (Binary Search) Trees
- Hash Maps
- Heaps
- Graphs



Binary Search Trees

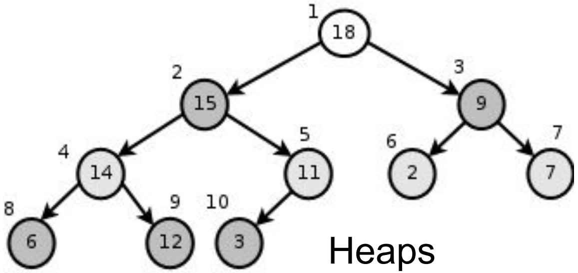


Linked Lists

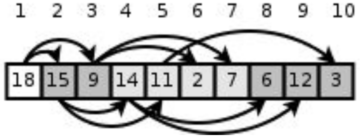


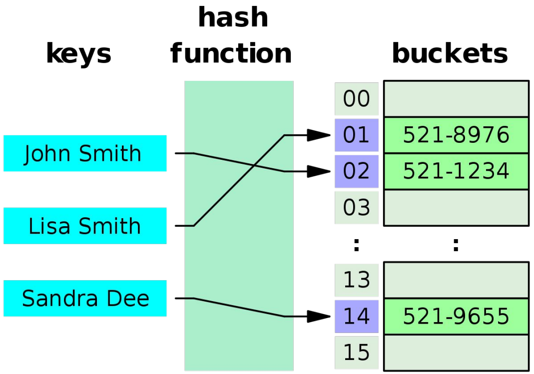
1

Arrays

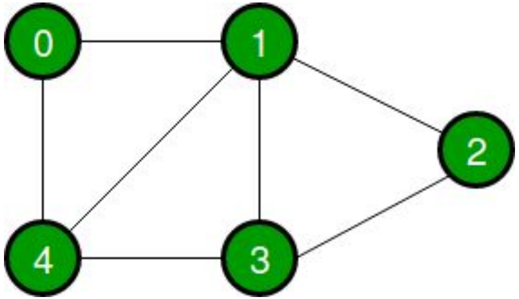


Heaps





Hash Maps



Undirected Cyclic Graph

Abstract Data Types (ADTs)

“Collection of data items, family of operations that operate on data.”

Usually limited to a **small set** of defined **operations**.
Can be implemented by a number of different **concrete data structures.**
But some might provide better **performance / time complexity** for the operations we want.

Examples: (usually implemented by)

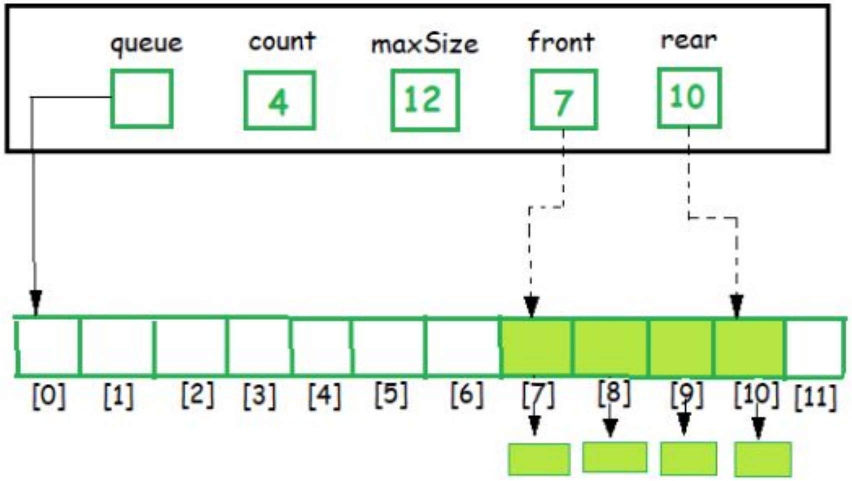
- List => array, linked list
- Stacks => array, singly-linked list
- Queues => singly / doubly linked list
- Priority Queues => heap, binary search tree
- Dictionary => hash map

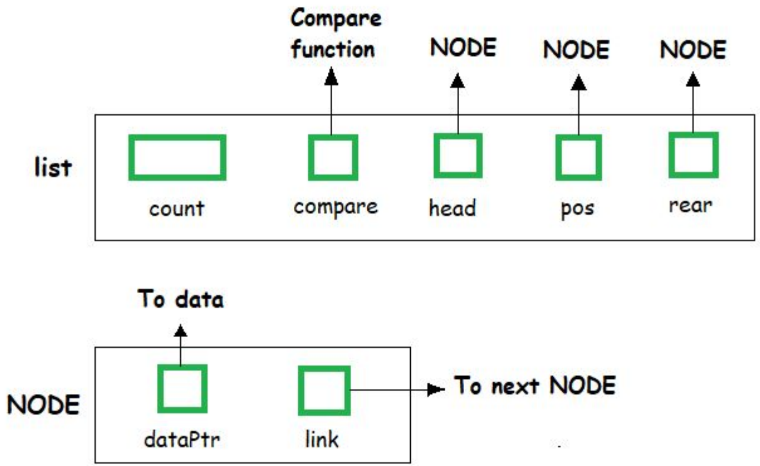
Lists (ADT)

- create empty list
- free/destroy the list
- is empty?
- add to start (head)
- add to end of list
- get first element of list (head)
- get all but first element of the list ("tail")
- get element at a specific index

Could be implemented by:

- Arrays (variable length / dynamic)
- Singly Linked List
- Doubly Linked List

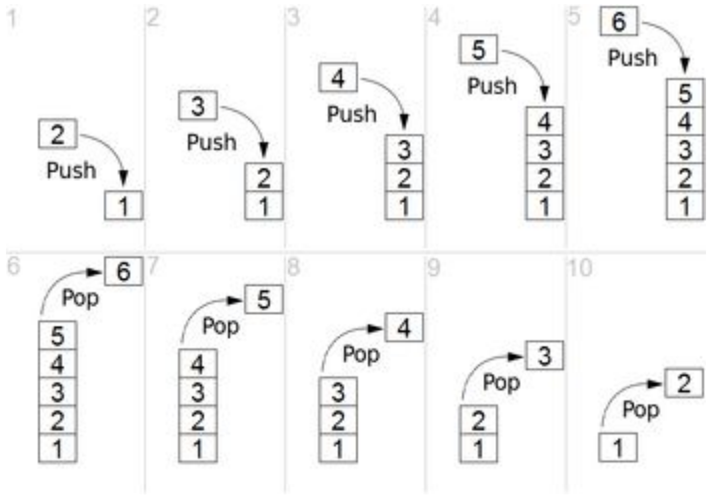


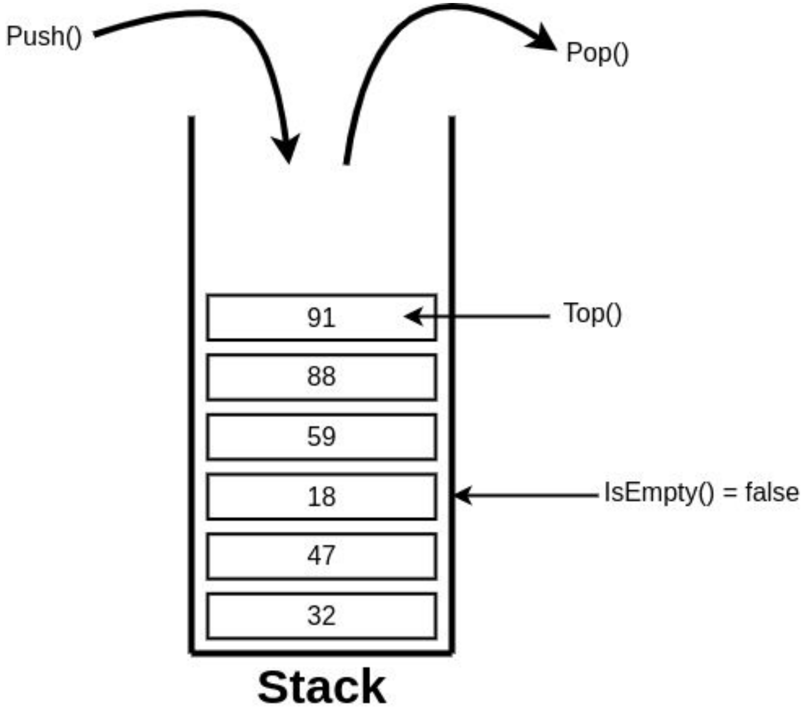


(GeeksForGeeks)

Stacks (ADT) **LIFO:** Last in First Out

- create new
- destroy / free
- is empty?
- push (add to top)
- pop (remove from top)
- top (get top element, but don't remove it)



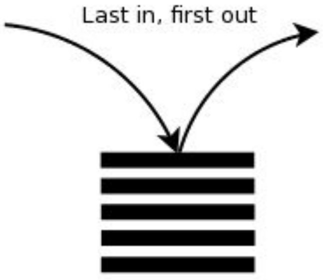


Stack Example

```
stack_t stack = new_stack();

push(stack, 5);
push(stack, 6);
push(stack, 4);
pop(stack);
pop(stack);
push(stack, 1);

printf(“%d”, pop(stack));

free_stack(stack);
```

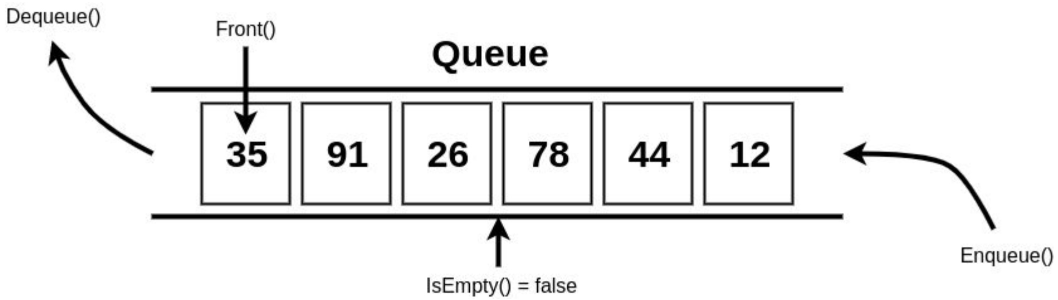
Stack Example

```
stack_t stack = new_stack();      // bottom -> {} <- top

push(stack, 5);                   // {5}
push(stack, 6);                   // {5, 6}
push(stack, 4);                   // {5, 6, 4}
pop(stack);                       // {5, 6}    4 popped
pop(stack);                       // {5}       6 popped
push(stack, 1);                   // {5, 1}

printf("%d", pop(stack));         // {5}       1 popped
                                  // 1

free_stack(stack);
```



Queues (ADT) **FIFO:** <u>First</u> in First Out

- create new
- destroy / free
- is empty?
- **enqueue** (add to end of queue)
- **dequeue** (remove from front of queue)
- front (get front, without removing it)

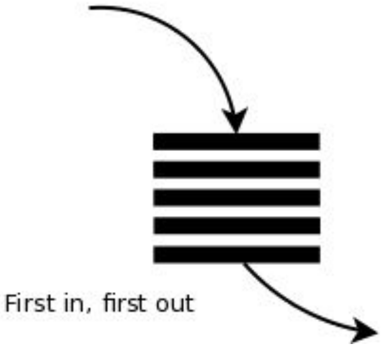


Queue Example

```
queue_t queue = new_queue();

enqueue(queue, 5);
enqueue(queue, 6);
enqueue(queue, 4);
dequeue(queue);
dequeue(queue);
enqueue(queue, 1);

printf(“%d”, dequeue(queue));

free_queue(queue);
```

Queue Example

```
queue_t queue = new_queue();        // front -> {} <- back

enqueue(queue, 5);                  // {5}
enqueue(queue, 6);                  // {5, 6}
enqueue(queue, 4);                  // {5, 6, 4}
dequeue(queue);                     // {6, 4}   5 popped
dequeue(queue);                     // {4}      6 popped
enqueue(queue, 1);                  // {4, 1}

printf("%d", dequeue(queue));       // {1}      4 popped
                    // 4
free_queue(queue);
```



Data Structures (Summary)

Data structures (DSs)
: Implementation ("C")
Calculate time complexity of operations

- Array
- Linked List
- Trees
- Hashmap
- Heap

Abstract data types (ADTs)
: Idea ("python")
Time complexity of operations depends on data structure used to implement it

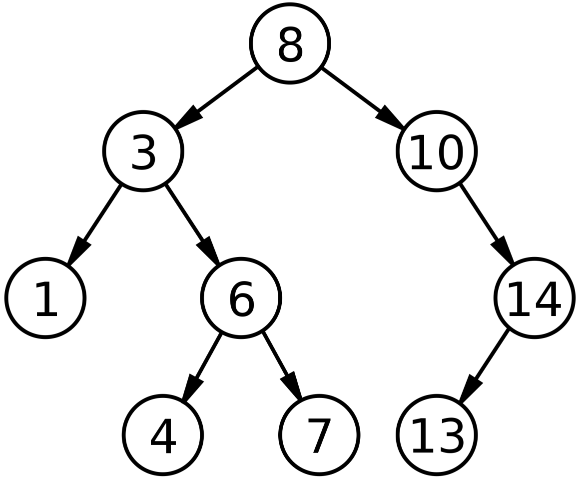
- List
- Stack
- Queue
- Dictionary
- Priority Queue

Binary Search Trees

Binary Search Tree

- Non-Linear **data structure**
- Binary tree: each **node** has up to **two** child nodes.
- Nodes are **linked** together using pointers (again)
- Structure keeps elements in sorted order, can always binary search.

```
typedef struct node node_t;
struct node {
    void *data; // polymorphic
    node_t *left;
    node_t *right;
};

typedef struct {
    node_t *root;
    int (*cmp)(void*,void*);
} tree_t;
```



cmp: arg1 is new item
arg2 is item in tree

Binary Search Tree

```
int int_cmp(void *a, void *b) { return *(int*)a - *(int*)b; }

int A[] = {4, 7, 2, 6, 9, 3, 1};
tree_t *tree = make_empty_tree(int_cmp);

for (int i = 0; i < 7; i++) insert_in_order(tree, A[i]);
```

Binary Search Tree

```
int int_cmp_incr(void *a, void *b) { return *(int*)a - *(int*)b; }

int A[] = {4, 7, 2, 6, 9, 3, 1};
tree_t *tree = make_empty_tree(int_cmp_incr);

for (int i = 0; i < 7; i++) insert_in_order(tree, A[i]);
```

Tree depth?

```
search_tree(tree, ____);  // is it in the tree?   O(  )?
```

Binary Search Tree

```
int int_cmp_incr(void *a, void *b) { return *(int*)a - *(int*)b; }

int A[] = {1, 2, 4, 6, 7, 8, 9};
tree_t *tree = make_empty_tree(int_cmp_incr);

for (int i = 0; i < 7; i++) insert_in_order(tree, A[i]);
```

Tree depth?

```
search_tree(tree, ____);  // is it in the tree?   O(   )?
```

Binary Search Trees

Useful for fast/insert/delete on sorted data.
NOT the same as binary search on sorted array!   -> O(log n) search!

Operation	**Average / best case (balanced)**	**Worst case (stick)**
Insert (ordered)	O(log n)	O(n)
Delete	O(log n)	O(n)
Search	O(log n)	O(n)

Design of Algorithms: Self-balancing binary trees red-black, 2-3, B-trees, AVL trees ...
https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree   O(log n) avg case!

Hashmaps: O(1)   average for insert, search, delete!

Arrays

Arrays, as a data structure

- Linear data structure -- items stored one after another
- Elements stored in contiguous memory locations.

1	6	9	4	5	7

In storing data, we have operations we want / need to perform.
For an array, what’s the complexity of these operations?

- Insert at front:
- Insert at end:
- Insert at random position:
- Delete at front:
- Delete at end:
- Delete at random position:
- Search for a value:

Arrays, as a data structure

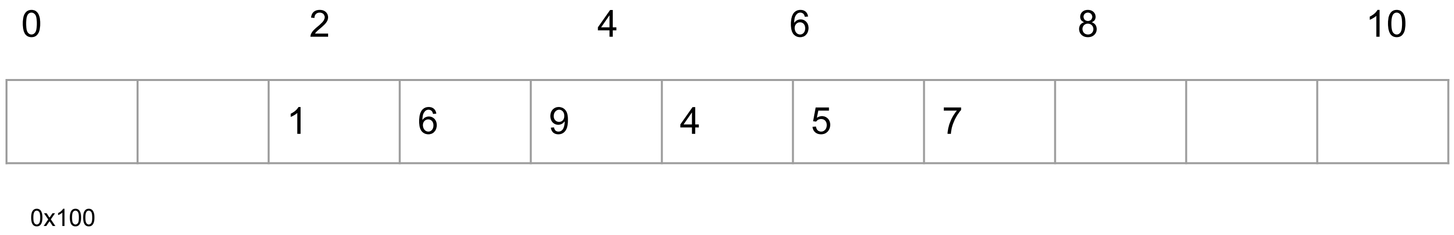
- Linear data structure -- items stored one after another
- Elements stored in contiguous memory locations.

1	6	9	4	5	7

In storing data, we have operations we want / need to perform.
For an array, what's the complexity of these operations?

- Insert at front: O(n) -- need to shift each element 1 over
- Insert at end: O(1), if there's room; otherwise O(n) to resize!
- Insert at random position: O(n) -- need to shift
- Delete at front: O(n) -- need to shift each element 1 back!
- Delete at end: O(1)
- Delete at random position: O(n) -- need to shift each element 1 back
- Search for a value: O(n) unsorted (linear), O(log n) sorted (binary)

capacity = 10
size = 6
start = 2   (or the memory address for it)

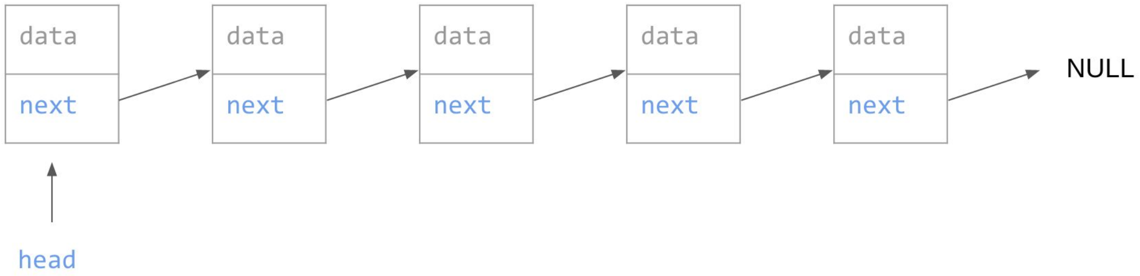


Linked List

Linked List

- Linear **data structure**
- Elements *not* stored in contiguous memory locations
- Elements are **linked** together using pointers

```
typedef int data_t;

typedef struct node node_t;

struct node {
    data_t data;
    node_t *next;
};

node_t *head = malloc(sizeof(node_t));
head->data = value;
head->next = NULL;
```



Linked List Operations

- Insert at head
- Remove at head
- Insert at tail
- Remove at tail

Draw it

Next slides for review.

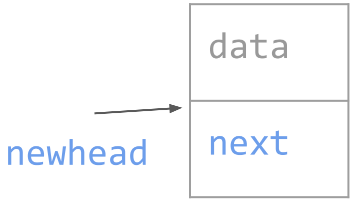
head = front = start
foot = end = tail

Note: if you are examined on coding this up, the question will guide you.
Just understand the concepts and complexities!

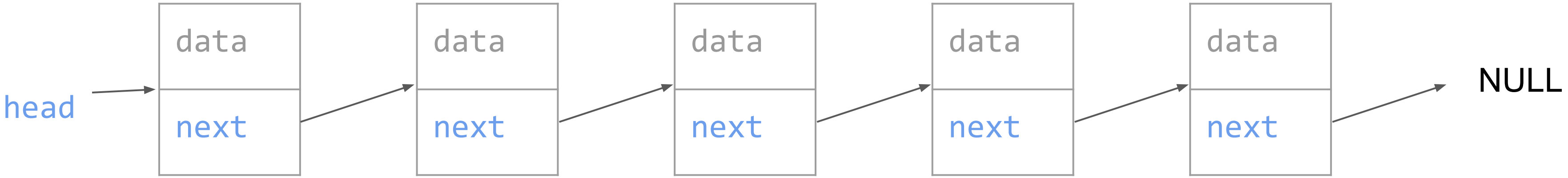
You’ll learn more about data structures and ADTs in COMP20007 Design of Algorithms.

Insert at head

`list_t *insert_at_head(list_t *list, data_t value)`



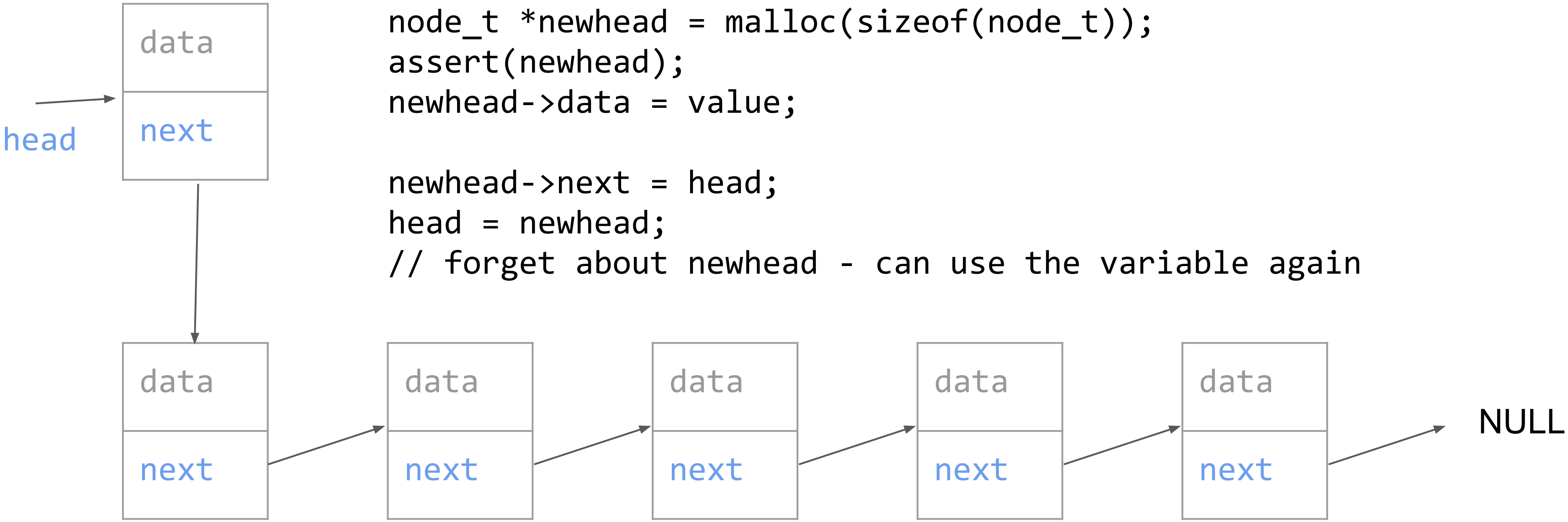

```
node_t *newhead = malloc(sizeof(node_t));
assert(newhead);
newhead->data = value;

...
```



Insert at head

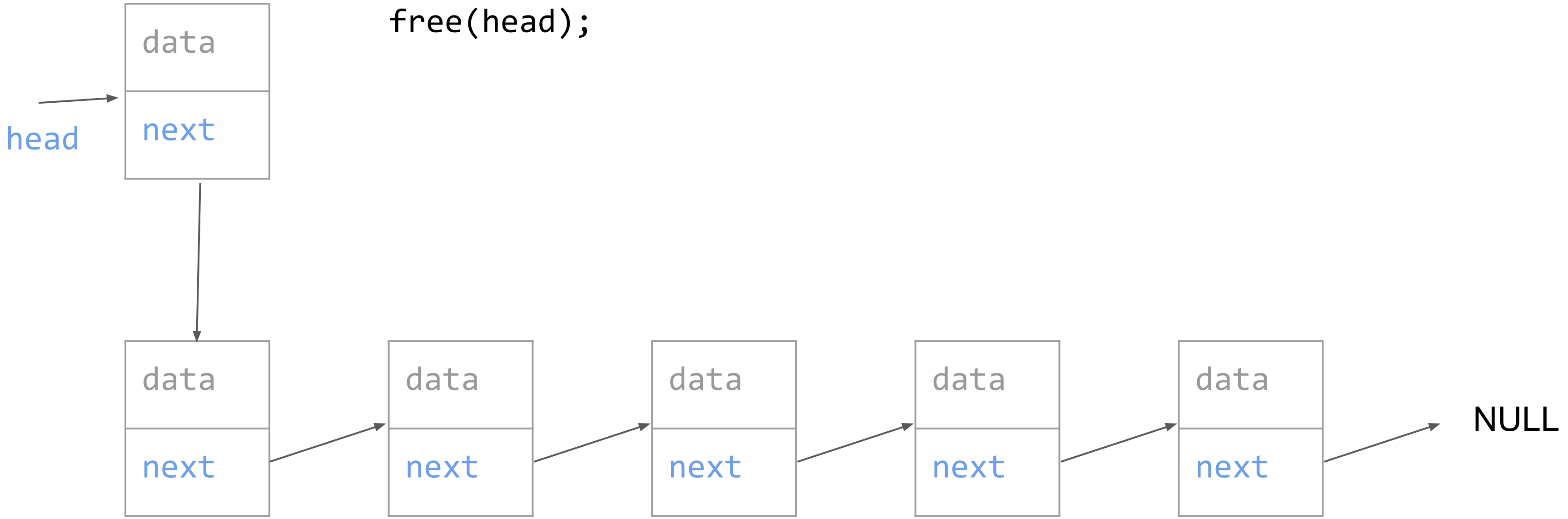
`list_t *insert_at_head(list_t *list, data_t value)`

```
node_t *newhead = malloc(sizeof(node_t));
assert(newhead);
newhead->data = value;

newhead->next = head;
head = newhead;
// forget about newhead - can use the variable again
```



Remove at head

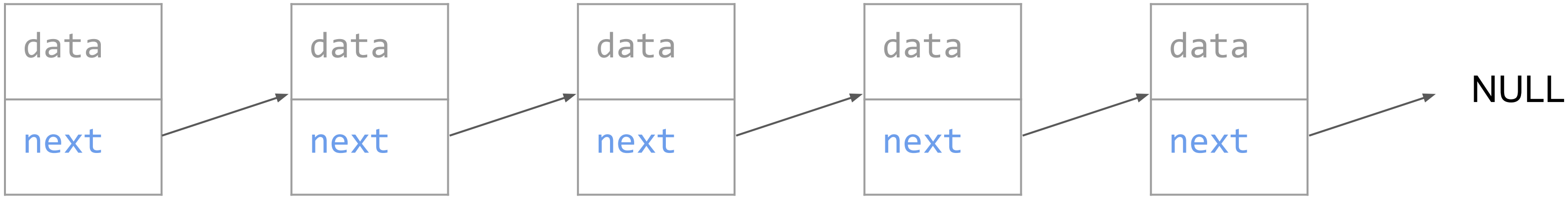
```
free(head);
```



Remove at head

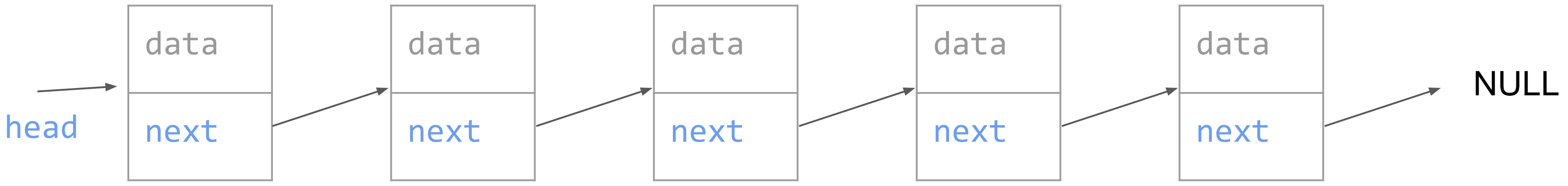
```
free(head);
```

There’s no pointer to
the new head!



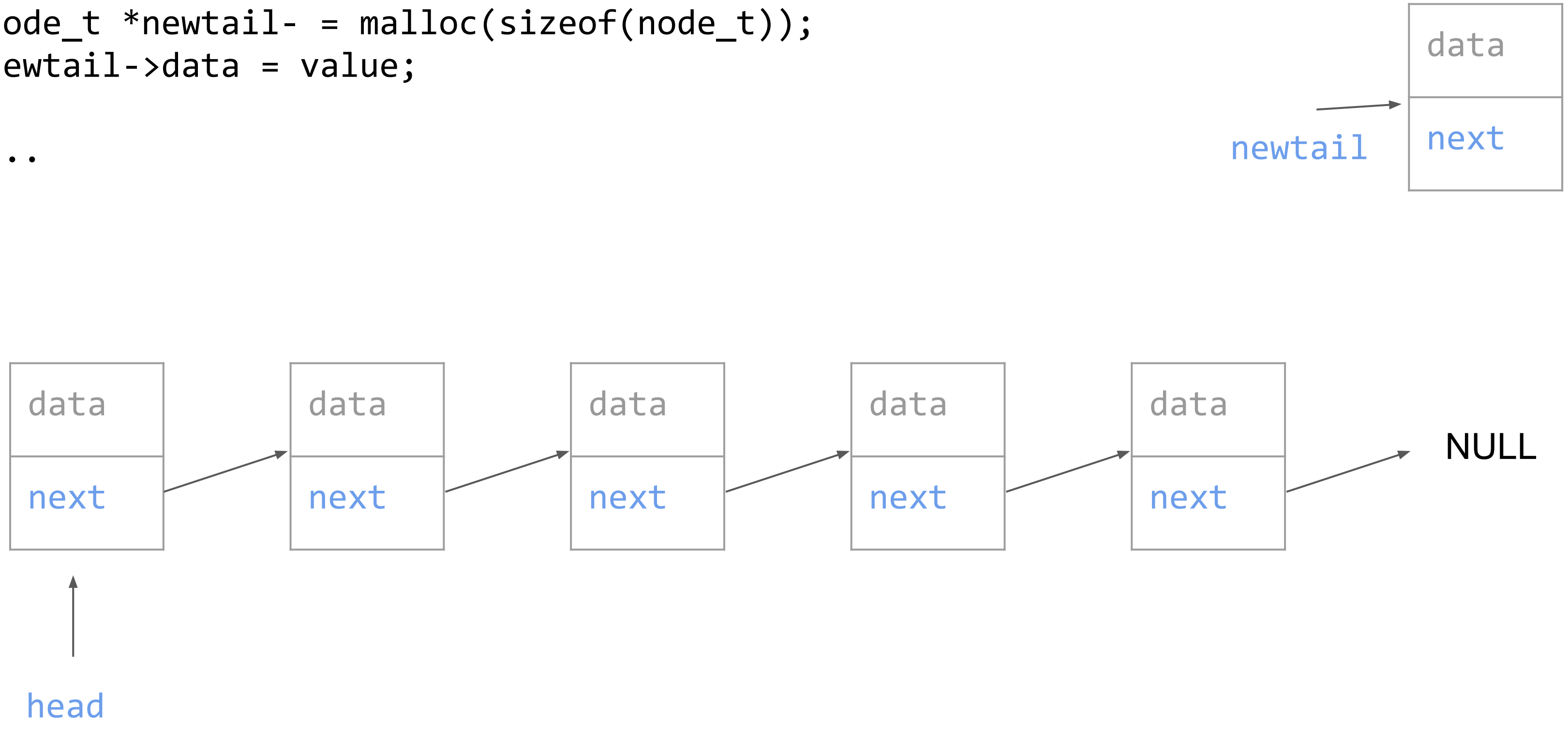
Remove at head

```
node_t *oldhead = head;
head = oldhead>next;
free(oldhead);
// oldhead = NULL;
```



Insert at tail

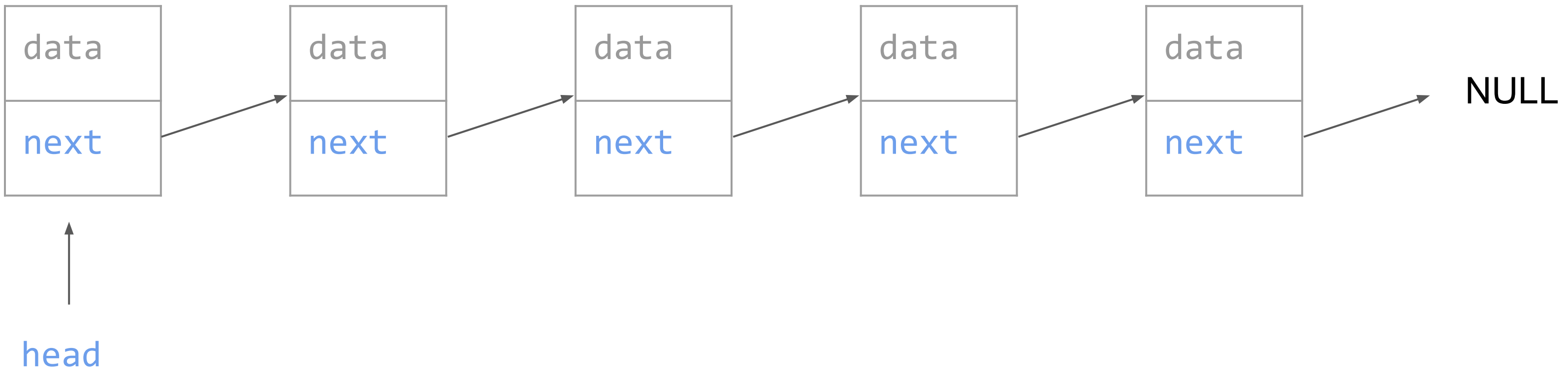
```
node_t *newtail- = malloc(sizeof(node_t));
newtail->data = value;

...
```



Linked list traversal (To tail)

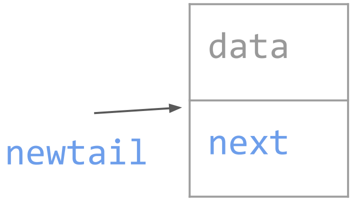
```
node_t *p = head;
assert(p);            // null->next is illegal
while (p->next) {
    p = p->next;
}
// p points to the tail.
```

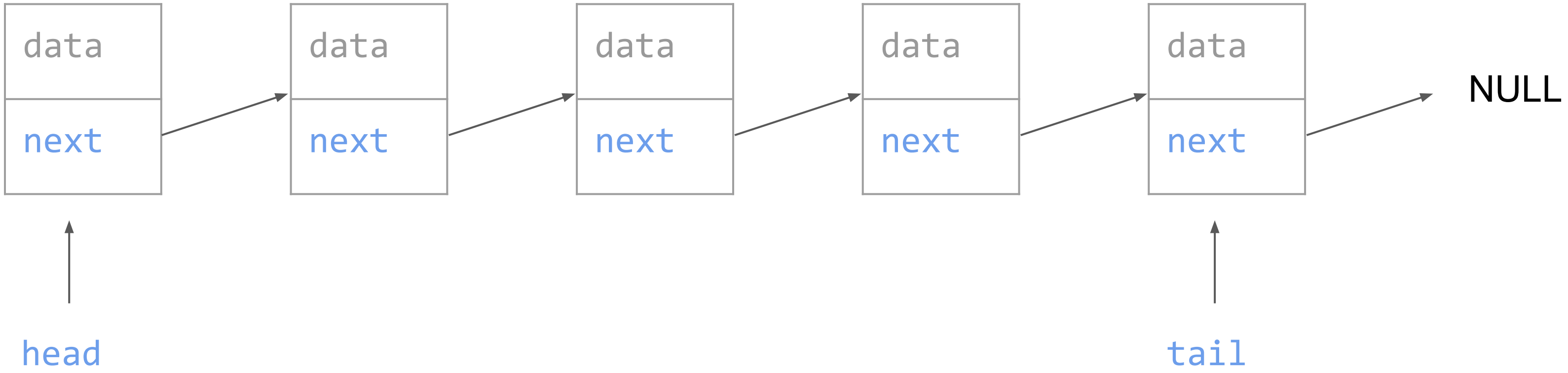
Works with head-tail and doubly as well.



Insert at tail

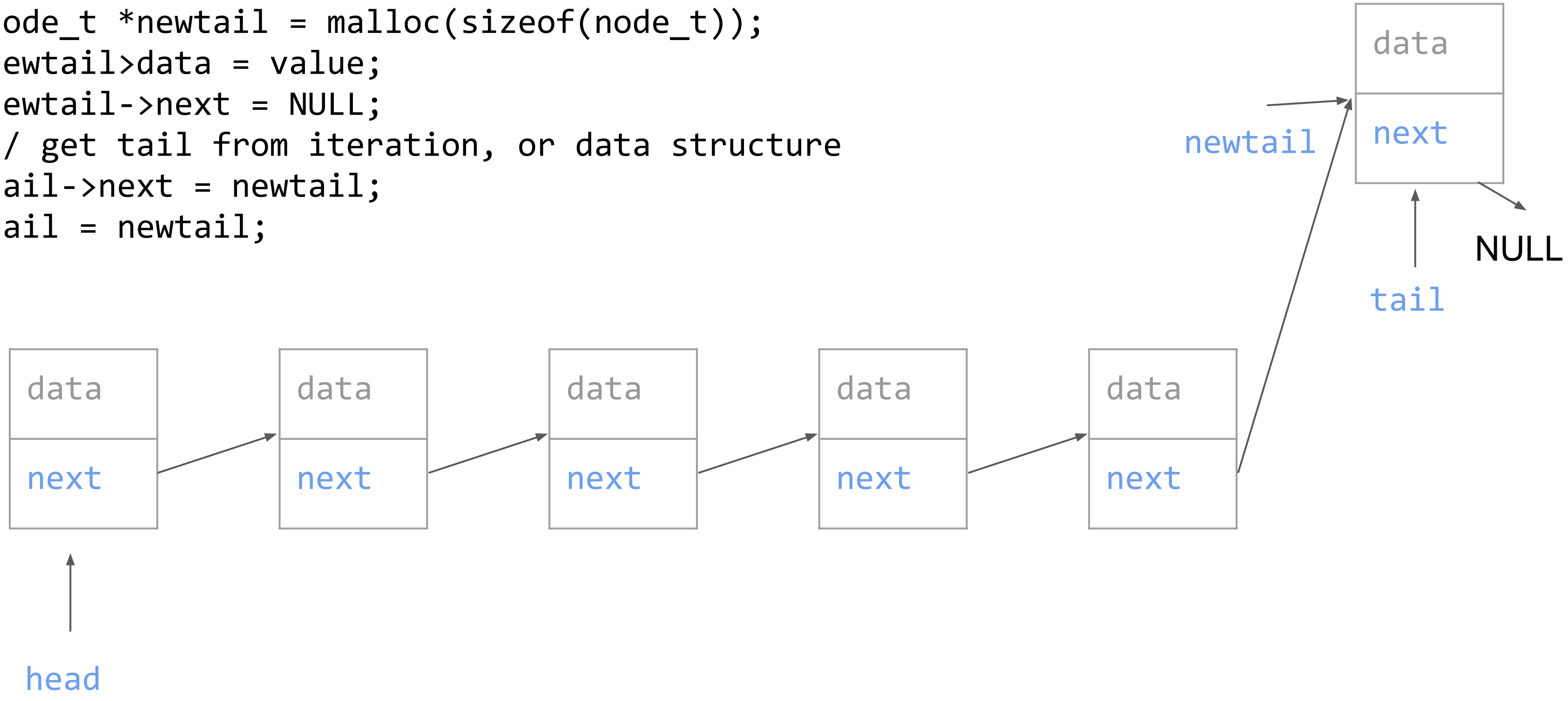
```
node_t *newtail = malloc(sizeof(node_t));
newtail>data = value;
newtail->next = NULL;
// get tail from iteration, or data structure
tail->next = newtail;
tail = newtail;
```





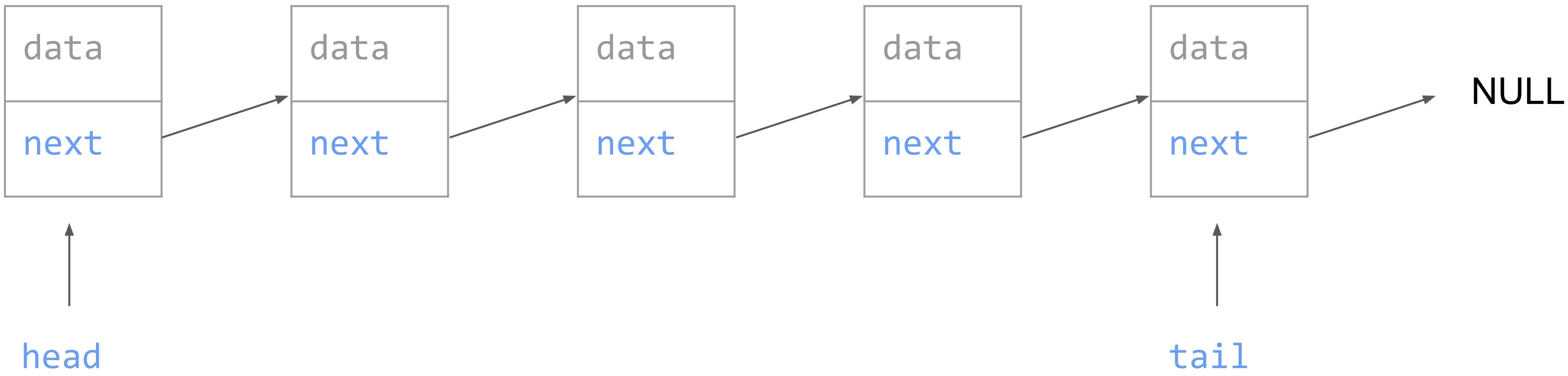
Insert at tail

```
node_t *newtail = malloc(sizeof(node_t));
newtail>data = value;
newtail->next = NULL;
// get tail from iteration, or data structure
tail->next = newtail;
tail = newtail;
```



Remove at tail

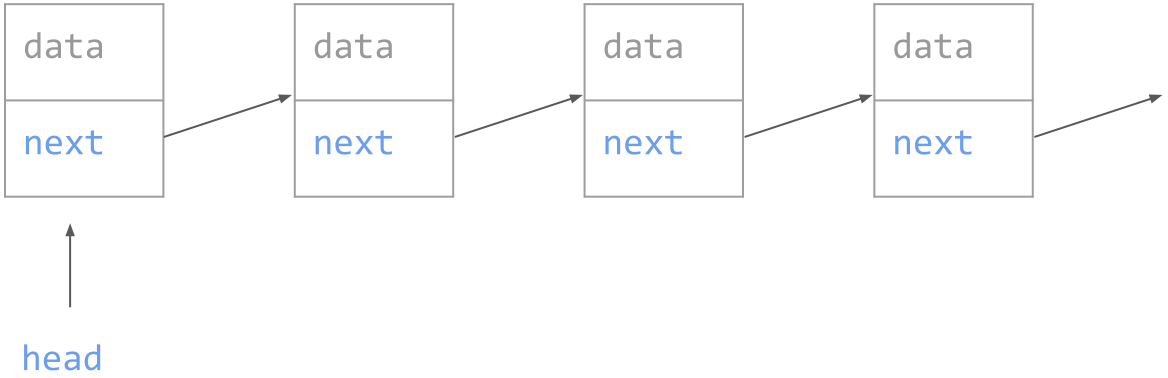
```
free(tail)?
```



Remove at tail

```
free(tail);
```

Need to set   second_last->next = NULL;
=> will be O(n) using a singly linked list.

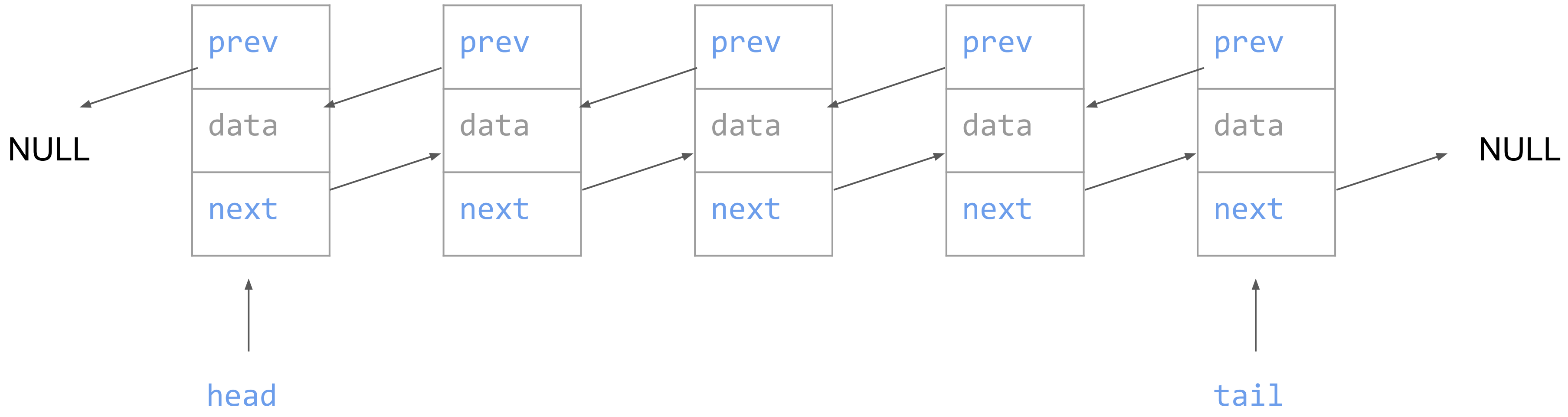


(garbage)

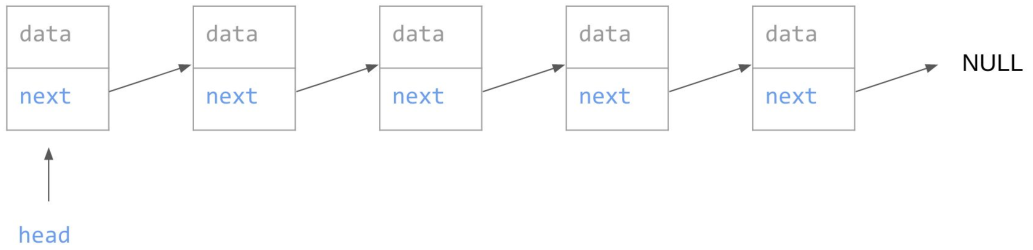
(it has a garbage collector)

Remove at tail

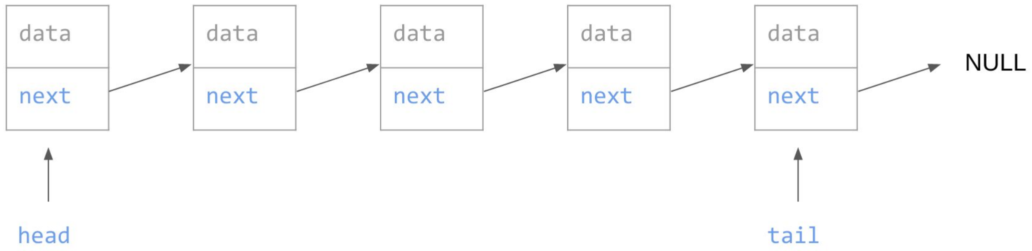
(WIP)



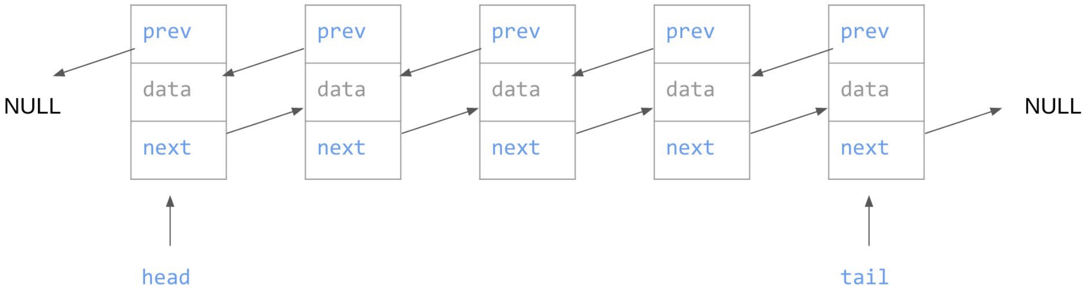
Linked List types



Singly Linked List (Head only)



Singly Linked List (Head & Tail)

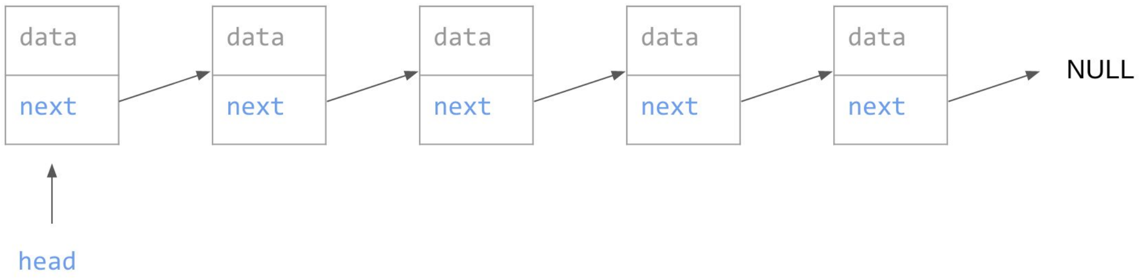


Doubly Linked List

Linked List types

Operation	Array	Singly head	Singly head/tail	Doubly
Insert at head	O(n)	O(1)	O(1)	O(1)
Delete at head	O(n)	O(1)	O(1)	O(1)
Insert at tail	O(1) or O(n) [resize]	O(n)	O(1)	O(1)
Delete at tail	O(1)	O(n)	O(n)	O(1)
Search for data	O(n) unsorted O(log n) sorted	O(n) avg/worst	O(n) avg/worst	O(n) avg/worst
space	n * sizeof(data) + 4 (store length)	n x sizeof(data) + 8(n+1)	n x sizeof(data) + 8(n+2)	n x sizeof(data) + 8(2n+2)
200 ints (size=4)	804	2048	2416	4016

Linked List (Singly, head only)



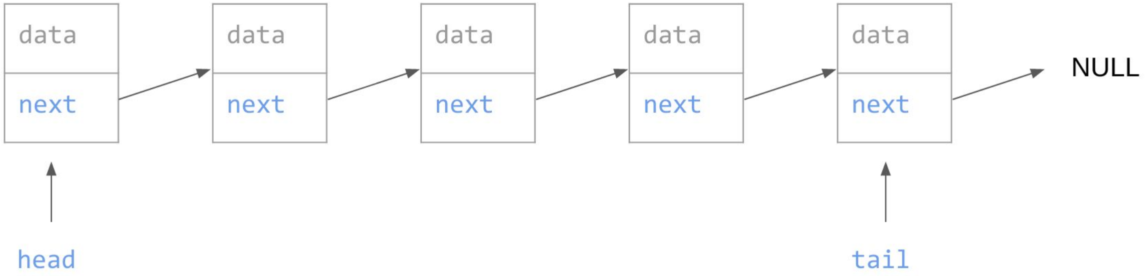

```
typedef int data_t;

typedef struct node node_t;

struct node {
    data_t data;
    node_t *next;
};

typedef node_t list_t; // “head”
```

Operation	**Complexity**
Insert at head	O(1)
Delete at head	O(1)
Insert at tail	O(n)
Delete at tail	O(n)
Search for data	O(1) best O(n) avg/worst

Space: n x sizeof(data)
+ 8 x n (next pointers)
+ 8 (head pointer)
= n x sizeof(data) + 8(n+1)

Can we improve?

Linked List (Singly, head & tail)




```
typedef int data_t;

typedef struct node node_t;
struct node {
    data_t data;
    node_t *next;
};
```

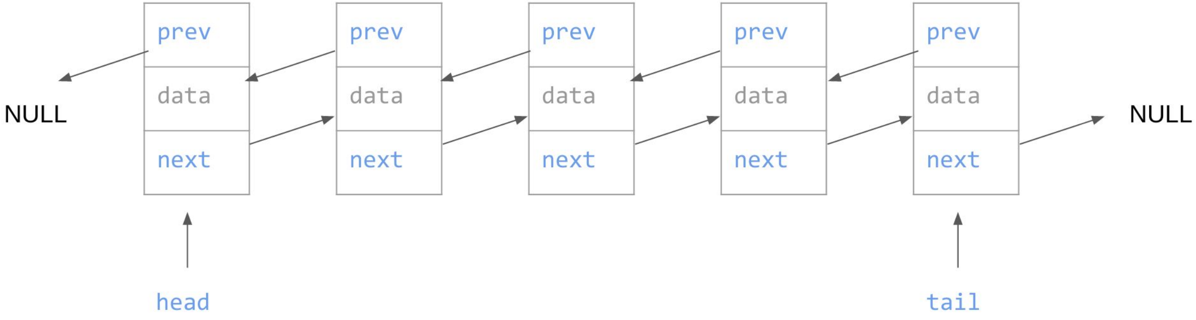
```
typedef struct {
    node_t *head;
    node_t *tail;
} list_t;
```

Operation	Complexity
Insert at head	O(1)
Delete at head	O(1)
Insert at tail	O(1)
Delete at tail	O(n)
Search for data	O(1) best O(n) avg/worst

Space: n x sizeof(data)
+ 8 x n (next pointers)
+ 8 x 2 (head, tail)
= n x sizeof(data) + 8(n+2)

Can we improve?

Doubly Linked List




```
typedef int data_t;

typedef struct node node_t;
struct node {
    data_t data;
    node_t *prev;
    node_t *next;
};
```

```
typedef struct {
    node_t *head;
    node_t *tail;
} list_t;
```

Operation	**Complexity**
Insert at head	O(1)
Delete at head	O(1)
Insert at tail	O(1)
Delete at tail	O(1)
Search for data	O(1) best O(n) avg/worst

Space: n x sizeof(data)
+ 8 x n x 2 (next, prev)
+ 8 x 2 (head, tail)
= n x sizeof(data) + 8(2n+2)

200 ints = 200x4 + 8x402 = **4016**

Array of 200 ints: 200x4 = **800**

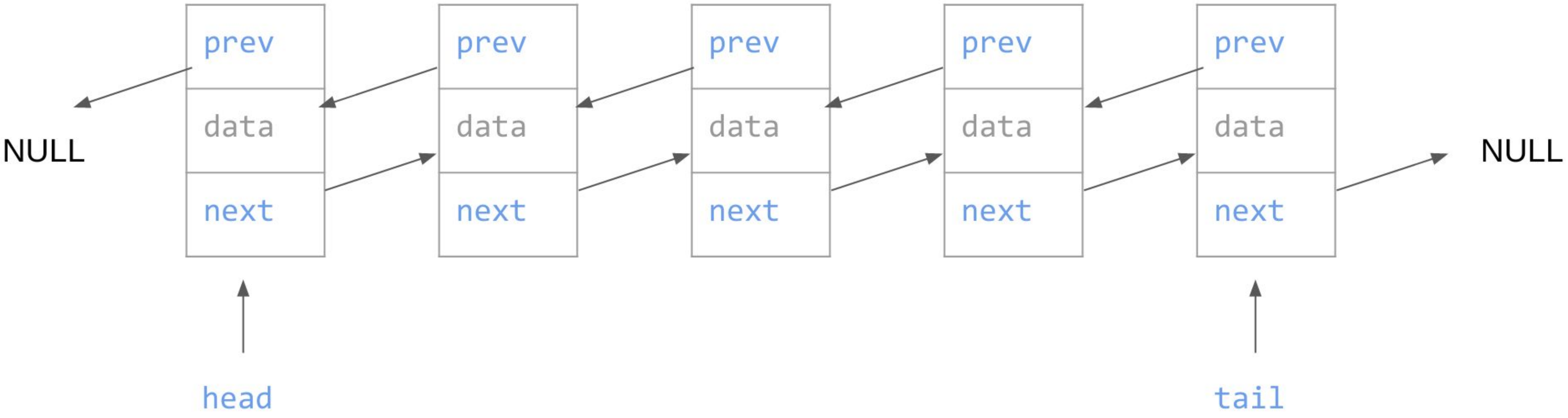
If the space complexity is so bad, why use it?

Space: n x sizeof(data)
+ 8 x n x 2 (next, prev)
+ 8 x 2 (head, tail)
= n x sizeof(data) + 8(2n+2)

200 ints = 200x4 + 8x402 = **4016**
Array of 200 ints: 200x4 = **800**

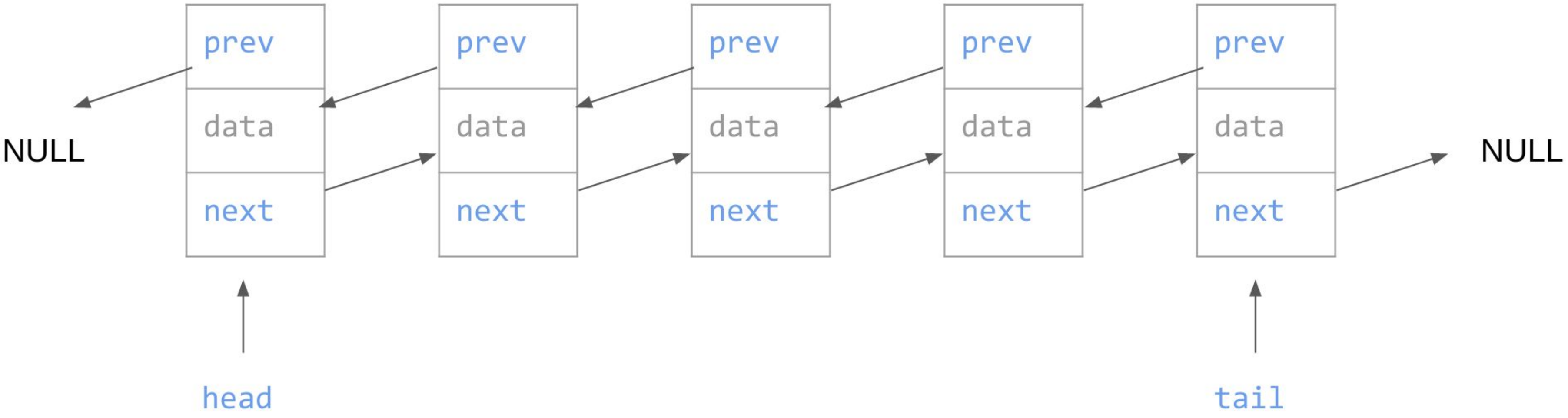
200 structs, sizeof(struct) = 5120 B (5 KB)
Linked list: 200x5120 + 8x402 = **1027216 B (~0.980 MB)**
Array: 200x5120 = **1024000 B (~0.977 MB)**

Searching Linked Lists



How to access the n/2th element?

Searching Linked Lists



How to access the n/2th element?
No **random access** in linked lists.
Array: A[n/2].

O(n) (ac, wc) for:

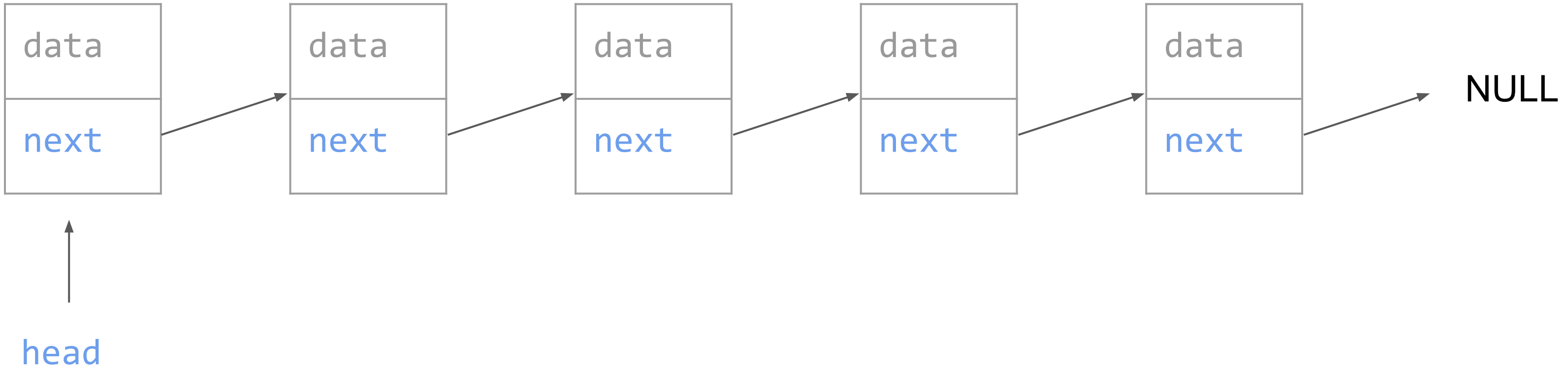
- Searching for an item in list
- Accessing / adding / replacing / removing item at random index

Linked list traversal (To kth element)

```
node_t *p = head;
assert(p);           // null->next is illegal
int i = 0;
while (p && i++ != k) {
    p = p->next;
}
// p points to kth element   (i - 1th element)
return p;
```

O(n)

Works with head-tail and doubly as well.

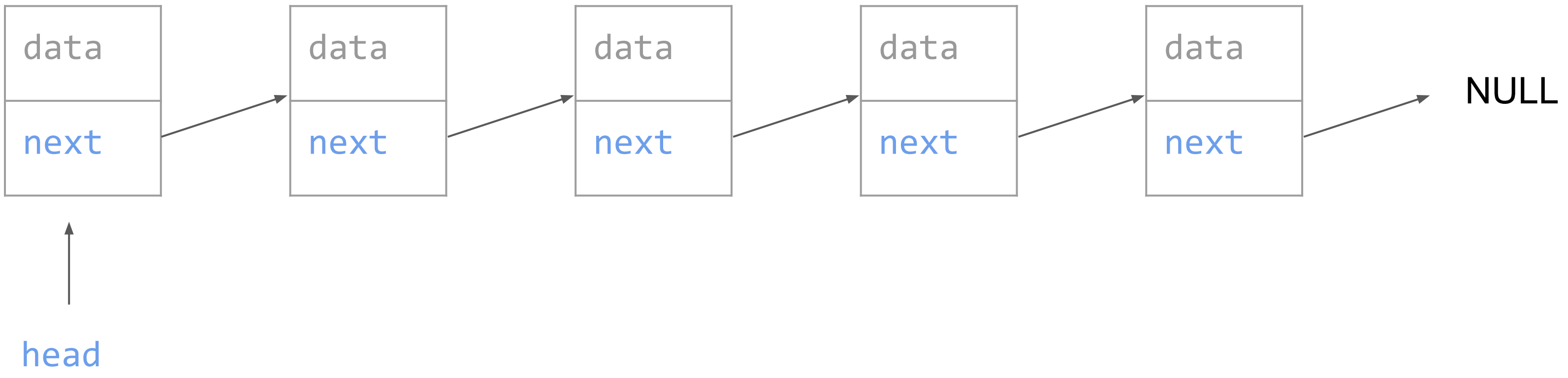


Linked list search

```
node_t *p = head;
assert(p);            // null->next is illegal
int i = 0;
while (p && p->data != target) {
    p = p->next;
}
// p points to target, or NULL.
return p;
```

O(n)

Works with head-tail and doubly as well.



Linked Lists vs Arrays Summary

Use arrays for:

- Randomly accessing items
- Binary search
- Quicksort
- Fast iteration

Use Linked Lists for:

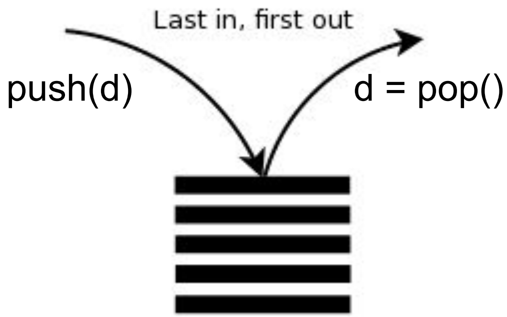
- Adding lists together
- Adding / removing items from start / end
- Mergesort

	Array	**Linked List (singly)**	**Doubly**
Insert at head	O(n) -- shifting	O(1) -- add at head	
Insert at tail	O(1) -- append O(n) sometimes (realloc)	O(n) -- need to traverse list to end (singly, head only) O(1) -- add at tail (singly, head & tail; or doubly)	
Insert random	O(n) -- shifting	O(n) -- no random access, traverse the list	
Delete at head	O(n) -- shifting	O(1) -- delete head	
Delete at tail	O(1)	O(n) -- need to update 2nd last element's next	O(1) -- update tail.prev
Delete random	O(n) -- shifting	O(n) -- no random access, traverse the list	
Replace / Access random	O(1) -- A[i] = x	O(n) -- no random access, traverse the list	
Search	O(n) linear (unsorted) O(log n) binary (sorted)	O(n) - can only search linearly	
Memory s = sizeof(data)	Less n x s	More n x s + 8 (n + (1 or 2))	Morer n x s + 8 (2n + 2)
Use for	Random access Fast sorted search	Fast insert/delete at head and tail No random access	

Stacks & Queues: Implementation

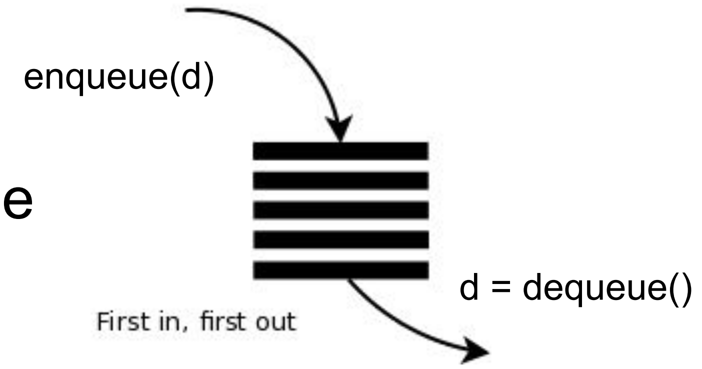
Stack

Last in, First out (LIFO)
push(data) -- add to top of stack
pop() -- remove from top of stack



Queue

First in, First out (FIFO)
enqueue(data) -- add to back of queue
dequeue() -- remove from front of queue



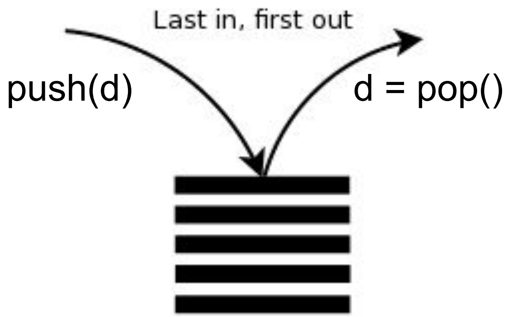
Stacks & Queues

Stack

Last in, First out (LIFO)

push(data) -- add to top of stack

pop() -- remove from top of stack

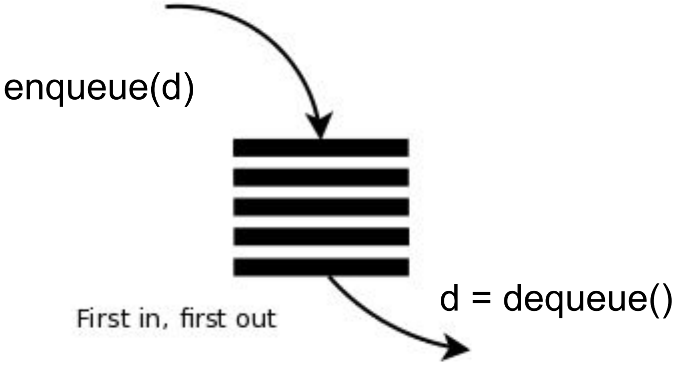


Queue

First in, First out (FIFO)

enqueue(data) -- add to back of queue

dequeue() -- remove from front of queue



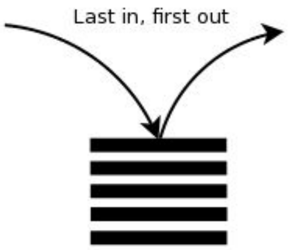
How to implement these ADTs with a data structure?
It would be nice to have these as O(1) operations.

Stack Implementation

Stack

Last in, First out (LIFO)
push(data) -- add to top of stack
pop() -- remove from top of stack

Which data structure gives optimal time complexity?



	Array	**Linked List; head only**	**head & tail**	**doubly**
Insert at head	O(n) -- shifting	O(1) -- traverse head		
Insert at tail	O(1) -- append O(n) sometimes	O(n) -- no tail pointer, need to traverse list.	O(1) -- traverse tail	
Delete at head	O(n) -- shifting	O(1) -- traverse head		
Delete at tail	O(1)	O(n) -- need to update 2nd last element's next		O(1) -- tail.prev
Memory (ints)	Best	Worse	Worser	Worst

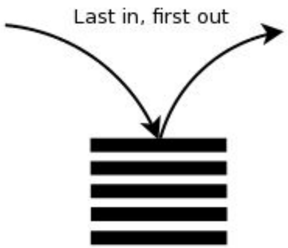
Stack Implementation

Stack

Last in, First out (LIFO)

push(data) -- add to top of stack (insert at head)

pop() -- remove from top of stack (delete at head)



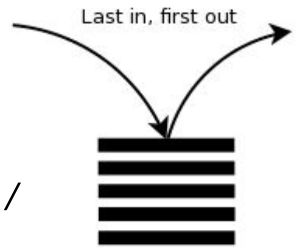
	Array	Linked List; head only	head & tail	doubly
Insert at head	O(n) -- shifting	O(1) -- traverse head		
Insert at tail	O(1) -- append O(n) sometimes	O(n) -- no tail pointer, need to traverse list.	O(1) -- traverse tail	
Delete at head	O(n) -- shifting	O(1) -- traverse head		
Delete at tail	O(1)	O(n) -- need to update 2nd last element's next		O(1) -- tail.prev
Memory (ints)	Best	Worse	Worser	Worst

Stack Implementation

Stack

Last in, First out (LIFO)
push(data) -- add to top of stack
pop() -- remove from top of stack

Arrays are fine for stacks of constant / maximum size



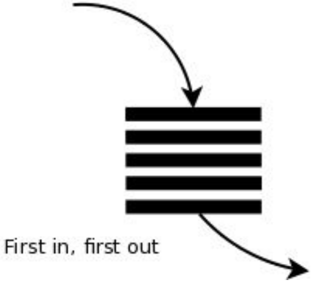
	Array	**Linked List; head only**	**head & tail**	**doubly**
Insert at head	O(n) -- shifting	O(1) -- traverse head		
Insert at tail	O(1) -- append O(n) sometimes	O(n) -- no tail pointer, need to traverse list.	O(1) -- traverse tail	
Delete at head	O(n) -- shifting	O(1) -- traverse head		
Delete at tail	O(1)	O(n) -- need to update 2nd last element's next		O(1) -- tail.prev
Memory (ints)	Best	Worse	Worser	Worst

Queue Implementation

Queue

First in, First out (FIFO)
enqueue(data) -- add to back of queue
dequeue() -- remove from front of queue

Which to use?



	Array	**Linked List; head only**	**head & tail**	**doubly**
Insert at head	O(n) -- shifting	O(1) -- traverse head		
Insert at tail	O(1) -- append O(n) sometimes	O(n) -- no tail pointer, need to traverse list.	O(1) -- traverse tail	
Delete at head	O(n) -- shifting	O(1) -- traverse head		
Delete at tail	O(1)	O(n) -- need to update 2nd last element's next		O(1) -- tail.prev
Memory (ints)	Best	Worse	Worser	Worst

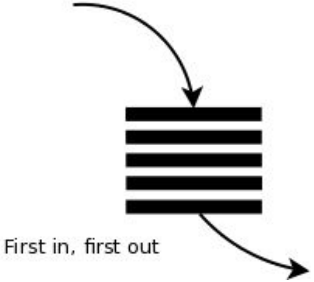
Queue Implementation

Queue

First in, First out (FIFO)

enqueue(data) -- add to back of queue (front of list)

dequeue() -- remove from front of queue (back of list)

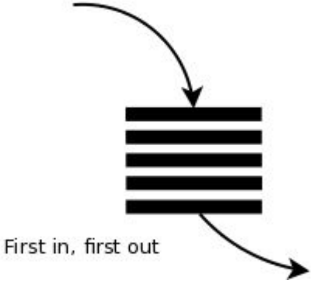


	Array	Linked List; head only	head & tail	doubly
Insert at head	O(n) -- shifting	O(1) -- traverse head		
Insert at tail	O(1) -- append O(n) sometimes	O(n) -- no tail pointer, need to traverse list.	O(1) -- traverse tail	
Delete at head	O(n) -- shifting	O(1) -- traverse head		
Delete at tail	O(1)	O(n) -- need to update 2nd last element's next		O(1) -- tail.prev
Memory (ints)	Best	Worse	Worser	Worst

Queue Implementation

Queue

First in, First out (FIFO)
enqueue(data) -- add to back of queue (back of list)
dequeue() -- remove from front of queue (front of list)



	Array	**Linked List; head only**	**head & tail**	**doubly**
Insert at head	O(n) -- shifting	O(1) -- traverse head		
Insert at tail	O(1) -- append O(n) sometimes	O(n) -- no tail pointer, need to traverse list.	O(1) -- traverse tail	
Delete at head	O(n) -- shifting	O(1) -- traverse head		
Delete at tail	O(1)	O(n) -- need to update 2nd last element's next		O(1) -- tail.prev
Memory (ints)	Best	Worse	Worser	Worst

Hashing

Try to implement the following **Abstract Data Type** using a Data Structure which makes each operation cost O(log n) in the average case.
Challenge: O(1) in the average case.

Dictionary {"Shaanan": "lecturer", "Tracy": "tutor", "Jianzhong": "coordinator"}
(key, value)

- add(key, value) - add a value with key to the dictionary (d[key] = value)
- get(key) - get the value stored with key (d[key])
- remove(key) - delete a key value pair (del d[key])

array -- O(n) add (linear search), O(n) remove, O(n) contains (linear search)
sorted array -- O(n) add (insertionsort), O(n) remove, O(log n) contains (binary search)
linked list -- O(n) add (linear search), O(n) remove, O(n) contains (linear search)
balanced BST O(log n) for everything! -sort by key

Hash Table -- O(1) for everything!

Dictionaries (ADT)

- create_new()
- **insert**(D, key, item) (want O(1))
- item <- **search**(D, key) (want O(1))
- **delete**(D, key) (want O(1))
- free()

```
{
    "Pride and Prejudice": "Alice",
    "Wuthering Heights": "Alice",
    "Great Expectations": "John"
}
```

Dictionaries (ADT)

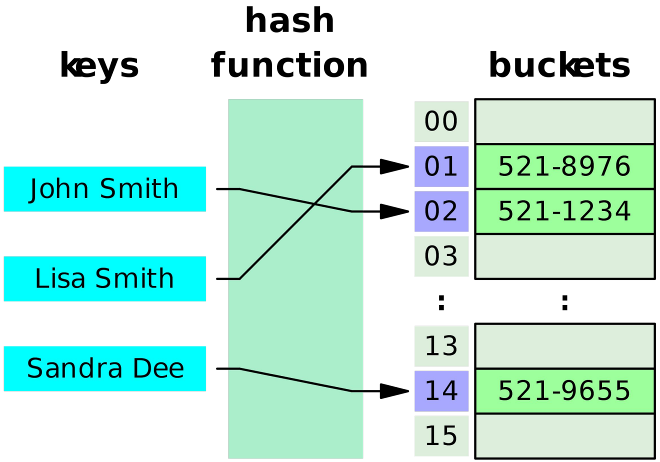
- create_new()
- **insert**(D, key, item) (want O(1))
- item <- **search**(D, key) (want O(1))
- delete(D, key) (want O(1))

```
{
    "Pride and Prejudice": "Alice",
    "Wuthering Heights": "Alice",
    "Great Expectations": "John"
}
```

Underlying data structure	Lookup		Insertion		Deletion		Ordered
	average	worst case	average	worst case	average	worst case	
Hash table	O(1)	O(n)	O(1)	O(n)	O(1)	O(n)	No
Self-balancing binary search tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
unbalanced binary search tree	O(log n)	O(n)	O(log n)	O(n)	O(log n)	O(n)	Yes
Sequential container of key–value pairs (e.g. association list)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	No

Hash tables / hash map

- Maps **keys** to **values**. (think python dictionary)
- Use a hash function to compute an **index** into an array of **buckets**
  - Ideally: each key is assigned into a unique bucket (**perfect hash function**)
  - We have **collisions**: hash function generates same index for >1 key.
  - There are ways to solve this!



	Average	Worst case
Insert	O(1)	O(n)
Search	O(1)	O(n)
Delete	O(1)	O(n)

Hash tables Issues

- **collisions**
  - hash function generates same index for multiple keys
  - can't store that element!
- **resizing**
  - run out of space in hashmap! => need to rehash everything => O(n)

h(x) = x%50
h(13) = 13
h(3) = 3
h(53) = 3

	Average	**Worst case**
Insert	O(1)	O(n)
Search	O(1)	O(n)
Delete	O(1)	O(n)

Load factor

$$\text{load factor} = \frac{n}{k}$$

```
n  is the number of entries occupied in the hash table

k  is the number of buckets
```

Higher load factor: greater chance of collision! (more buckets are full)
Lower load factor: wasted memory, and not necessarily any reduction in search cost

Java: aims for load factor of 0.75.
If the HashMap's load reaches 0.75, we **resize** the hashmap.

Resizing: double array size, need rehash all values => O(n) … bad!

Collision resolution

The way we handle collisions will erode the performance of the hash table.
We lose O(1)!

- Separate chaining
- Cuckoo hashing
- Linear probing

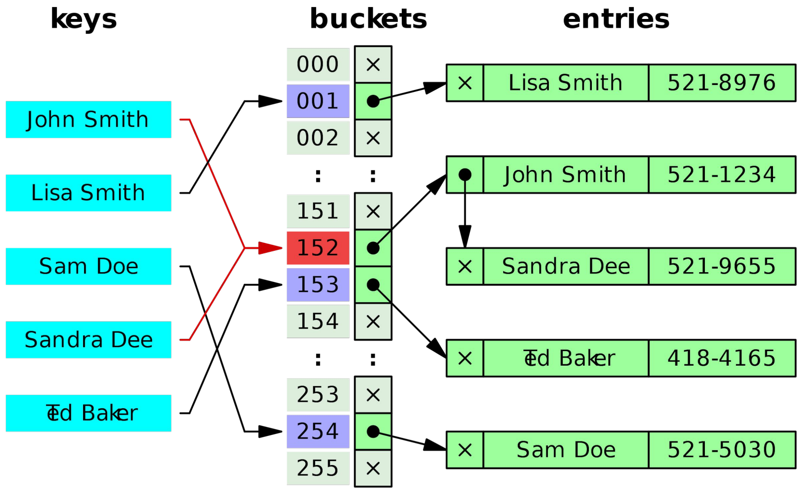
To consider: do we maintain O(1) insert/delete/search with these schemes?

Separate chaining

Instead of having buckets of values, have buckets of some **secondary data structure.** (eg: linked list, array, another hashmap => all different properties)

- Each bucket has a list of entries with the same index.
- O(1) to find the correct bucket, then O(m) to search through the bucket:

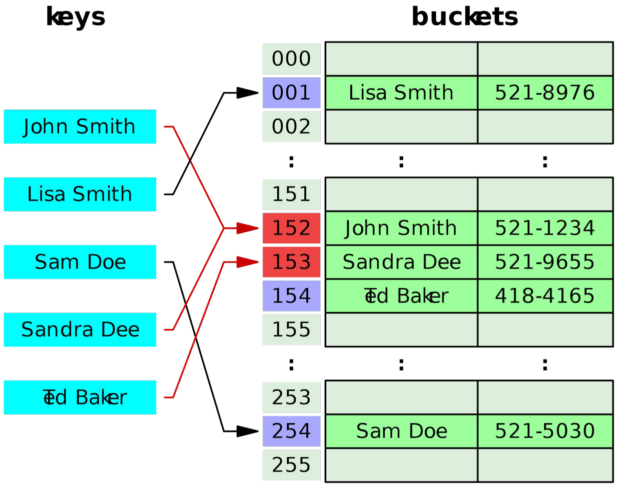
Searching through linked list: O(n)
Balanced binary tree: O(log n)
    HashMap Java 8



Linear Probing

- Buckets are values as usual.
- If there is a collision, we **search linearly forward** until an unoccupied slot is found. (eg: idx 100 full? go to 101. or 102. or 103…)

Time complexity: O(n) still, just resolves the collision part.
- How to delete elements? O(n) search. -- bad!



Cuckoo hashing

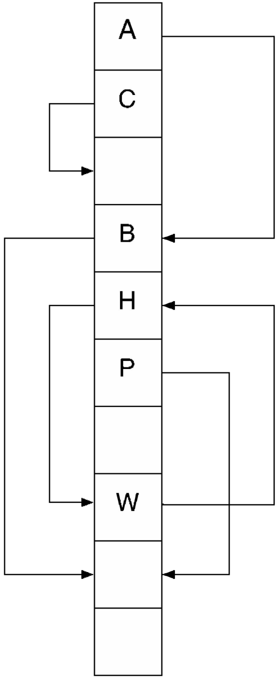
- Use multiple hash functions.
- Lookup: use hash function 1
  - If not found, use hash function 2
  - Etc
- Insertion: use hash function 1.
  - If collision, use hash function 2
  - Etc
- Deletion, same as lookup.

Worst case: O(1)

Careful with cycles!

$$h(6) = 6 \mod 11 = 6$$

$$h'(6) = \left\lfloor \frac{6}{11} \right\rfloor \mod 11 = 0$$

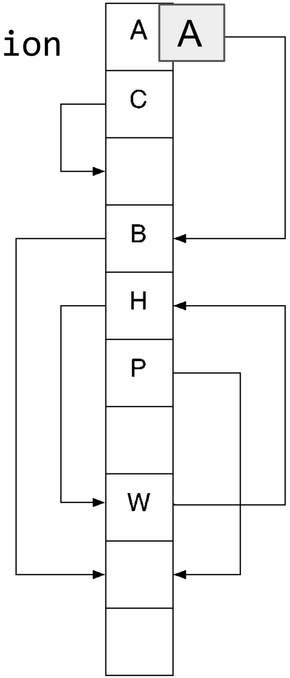


Cuckoo hashing

```
on collision: displace the previous object to allow the new one to
be in its right spot, then re-insert it using a different hash function

  -   Use multiple hash functions.
  -   Lookup: use hash function 1
        -   If not found, use hash function 2.
  -   Insertion: use hash function 1.
        -   If collision, use hash function 2
  -   Deletion, same as lookup.
Worst case: O(1)
Efficient; but careful with cycles!
```

insert A:
h(A) = 4   (H[4]=B)
h'(A) = 8 (H[8]=/)
H[8] := A

insert A:
h(A) = 4   (H[4]=B)
h'(B) = 8 (H[8]=/)
H[8] := B



$$h(6) = 6 \mod 11 = 6$$

$$h'(6) = \left\lfloor \frac{6}{11} \right\rfloor \mod 11 = 0$$

6 hashes to itself...

Issues with hashing

- Collisions!
  - The way we handle collisions will erode the performance of the hash table. Loses O(1)!
  - Want to avoid collisions.

- If the hash table is nearly full -> more likely to have a collision!
  - If array is full -> guaranteed for collision!

- If the hash table is nearly empty -> wasting space!
  - Similar to having an array of size 1000.
    We want to start small and grow.

Solution:
If hash table is at <u>75%</u> capacity - resize it.
How to resize?
    Double the size of the array, AND need to **rehash** every element. Time?

Hash tables / hash map

- Maps **keys** to **values**.   (think python dictionary)
  - in memory, same as an array ("associative array")

- Use a <u>hash function</u> to compute an **index** into an array of **buckets**
- Ideally: each key is assigned into a unique bucket (**perfect hash function**)
  - eg: hash ints: h(x) = x % 10   (set N_BUCKETS = k = 10)

% modulo (remainder)

k=10 n=0

Insert into hash table: ~~2~~, 7, 85, 8, 28, 6.

13 / 10 = remainder? 3

k: # buckets, n: # elements in map

H			2	13		85		7	8 |28	
idx	0	1	2	3	4	5	6	7	8	9

Hash tables / hash map

- Use a <u>hash function</u> to compute an **index** into an array of **buckets**
- Ideally: each key is assigned into a unique bucket (**perfect hash function**)
  - eg: hash ints: h(x) = x % N_BUCKETS (set N_BUCKETS = k = 10)

```
h1(x) = x % 10;       k: # buckets;  n = num elements
h2(x) = x % 20;     chance of collision: 1/(k - n)
```

Insert into hash table: 2, 85, 8, 28, 6, 17. h(28) = 28 % 10 = 8

2 5 8 8 6 7 17

			2			85	6		8 28	
H								17		

idx	0	1	2	3	4	5	6	7	8	9