CPS 305

Data Structures

Prof. Alex Ufkes

Topic 5: Unconventional Sorting, Linked Lists in LISP

Toronto Metropolitan University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

Previously: Comparison Sorting

Sorting algorithms based on Boolean comparison of pairs of elements

Requirements:

1. If a `<=` b, and b `<=` c, then a `<=` c
2. For all a and b, either a `<=` b or b `<=` a

$O(n^2)$: Selection sort, Insertion sort, Bubble sort
`O(nlogn)`: Merge sort, Quick sort

Moving On…

Unconventional Sorting Algorithms

Comparison Sorting: Alternatives?

Are there sorting algorithms based on something ***other*** than comparison?

There are, but they are less common:

- They don’t generalize across all types (only for integers, strings)
- They only perform well when data is uniformly distributed
- And more

We’ll see three:

- Counting sort
- Radix sort
- Bucket sort

Counting Sort

- Keys (value to sort over) are small integers
- How small? They're used as array indexes.

Counting Sort: Part 1

Iterate through array, compute a histogram of occurrences:

i	0	1	2	3	4	5	6	7	8	9
count	0	0	0	0	0	0	0	0	0	0
keys	0	0	1	2	5	4	2	7	2	5

i	0	1	2	3	4	5	6	7	8	9
count	0	0	0	0	0	0	0	0	0	0
keys	0	0	1	2	5	4	2	7	2	5

Before we get started:

- The size of `count` must be `max(keys)+1`
- Thus, if `keys` contains very large values, `count` will take up a lot of space.
- Here, the max value in `keys` is 7, but for the sake of demonstration we'll just keep `count` and `keys` the same size.

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	1	0	0	0	0	0	0	0	0	0
keys	0	0	1	2	5	4	2	7	2	5

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	2	0	0	0	0	0	0	0	0	0
keys	0	0	1	2	5	4	2	7	2	5

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	2	1	0	0	0	0	0	0	0	0
keys	0	0	1	2	5	4	2	7	2	5

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	**2**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	2	1	1	0	0	1	0	0	0	0
keys	0	0	1	2	5	4	2	7	2	5

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	**2**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	2	1	2	0	1	1	0	0	0	0
keys	0	0	1	2	5	4	2	7	2	5

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	2	1	2	0	1	1	0	1	0	0
keys	0	0	1	2	5	4	2	7	2	5

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	2	1	3	0	1	1	0	1	0	0
keys	0	0	1	2	5	4	2	7	2	5

Counting Sort: Part 1

i	0	1	2	3	4	5	6	7	8	9
count	2	1	3	0	1	2	0	1	0	0
keys	0	0	1	2	5	4	2	7	2	5

Done!

Counting Sort: Part 2

i	0	1	2	3	4	5	6	7	8	9
count	2	1	3	0	1	2	0	1	0	0
keys	0	0	1	2	5	4	2	7	2	5

- Next, we perform a ***prefix sum*** computation.
- This determines, for each key, the starting position in the array for items having that key.

Counting Sort: Part 2

i	0	1	2	3	4	5	6	7	8	9
count	2	1	3	0	1	2	0	1	0	0

- This is essentially a running sum, offset by one index.
- In practice we'd just overwrite **count**.
- We'll use another helper array for clarity.

Counting Sort: Part 2

i	0	1	2	3	4	5	6	7	8	9
count	2	1	3	0	1	2	0	1	0	0
pref	0	2	3	6	6	7	9	9	10	10

Counting Sort: Part 2

i	0	1	2	3	4	5	6	7	8	9
`count`	2	1	3	0	1	2	0	1	0	0
`pref`	0	2	3	6	6	7	9	9	10	10

In practice we'd just overwrite count.

Counting Sort: Part 2

i	0	1	2	3	4	5	6	7	8	9
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**
count	**0**	**2**	**3**	**6**	**6**	**7**	**9**	**9**	**10**	**10**

We can now produce the sorted output array

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
`keys`	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**
`count`	**0**	**2**	**3**	**6**	**6**	**7**	**9**	**9**	**10**	**10**

We loop over **keys**, placing elements into an output array using `count` to determine where each element should go.

	0	1	2	3	4	5	6	7	8	9
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**
count	**0**	**2**	**3**	**6**	**6**	**7**	**9**	**9**	**10**	**10**
output										

- `count` at index **k** tells us where the ***first*** key with value k should go.
- **(**`count` at index **k)+1** tells us where the ***next*** key with value k should go.
- We increment elements in count as we place keys in the output array.

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	0	0	1	2	5	4	2	7	2	5
count	1	2	3	6	6	7	9	9	10	10
output	0									

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**
count	**2**	**2**	**3**	**6**	**6**	**7**	**9**	**9**	**10**	**10**
output	**0**	**0**								

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**
count	**2**	**3**	**3**	**6**	**6**	**7**	**9**	**9**	**10**	**10**
output	**0**	**0**	**1**							

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	0	0	1	2	5	4	2	7	2	5
count	2	3	4	6	6	7	9	9	10	10
output	0	0	1	2						

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	0	0	1	2	5	4	2	7	2	5
count	2	3	4	6	6	8	9	9	10	10
output	0	0	1	2				5		

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	0	0	1	2	5	4	2	7	2	5
count	2	3	4	6	7	8	9	9	10	10
output	0	0	1	2			4	5		

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**
count	**2**	**3**	**5**	**6**	**7**	**8**	**9**	**9**	**10**	**10**
output	**0**	**0**	**1**	**2**	**2**		**4**	**5**		

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	0	0	1	2	5	4	2	7	2	5
count	2	3	5	6	7	8	9	10	10	10
output	0	0	1	2	2		4	5		7

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	0	0	1	2	5	4	2	7	2	5
count	2	3	6	6	7	8	9	10	10	10
output	0	0	1	2	2	2	4	5		7

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	0	0	1	2	5	4	2	7	2	5
count	2	3	6	6	7	9	9	10	10	10
output	0	0	1	2	2	2	4	5	5	7

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**
count	**2**	**3**	**6**	**6**	**7**	**9**	**9**	**10**	**10**	**10**
output	**0**	**0**	**1**	**2**	**2**	**2**	**4**	**5**	**5**	**7**

Counting Sort: Part 3

	0	1	2	3	4	5	6	7	8	9
keys	**0**	**0**	**1**	**2**	**5**	**4**	**2**	**7**	**2**	**5**
output	**0**	**0**	**1**	**2**	**2**	**2**	**4**	**5**	**5**	**7**

- We’ve iterated through keys twice, and count once
- Thus, the complexity of counting sort is **`O(3n)=O(n)`**
- However! Remember length of count is max(keys)
- Notice also that counting sort is **stable**!

Radix Sort

Counting sort can be used as a subroutine in Radix sort

- “Radix” refers to the base of the number.
- Most used on strings and integers but can operate on any arbitrary bit pattern.
- Sort integers in similar fashion as counting sort
- Hierarchically group keys by their individual digits
- **Two flavors:** LSD and MSD

Radix Sort

Two flavors: LSD and MSD

Least significant digit (LSD)

- Sort values based on their right-to-left digit order

7389566

- Good for sorting in numeric order
- Short keys (smaller numbers) come before large keys.

Radix Sort

Two flavors: MSD and LSD

Most significant digit (MSD)

- Sort values based on their left-to-right digit order

```
asdfghj
```

- Sort based on ***lexicographic*** order
- Good for sorting strings
- Doesn't sort in numeric order, 10 would come before 3.

Radix Sort

10	8	172	15	44	3	212	888	20	15

Let **p** = max number of digits per key = **3**
Let **r** = radix (base) of keys = **10**

Create **r** buckets, sort elements into buckets based on digit currently being evaluated. Total number of passes = **p**.

LSD

10	8	172	15	44	3	212	888	20	15

0	1	2	3	4	5	6	7	8	9
10 20		172 212	3	44	15 15			8 888	

LSD

10		**172**	**3**	**44**	**15**			**8**	
20		**212**			**15**			**888**	
0	1	2	3	4	5	6	7	8	9

Repopulate array with this intermediate result:

LSD

10	**20**	**172**	**212**	**3**	**44**	**15**	**15**	**8**	**888**

Perform binning operation again, this time looking at the 2^{nd} LSD
We will add leading zeros to make the process more clear

LSD

010	020	172	212	003	044	015	015	008	888

0	1	2	3	4	5	6	7	8	9
003	010	020		044			172	888	
008	212								
	015								
	015								

LSD

003	**010**	**020**		**044**			**172**	**888**	
008	**212**								
	015								
	015								
0	1	2	3	4	5	6	7	8	9

Repopulate array with this intermediate result:

LSD

003	**008**	**010**	**212**	**015**	**015**	**020**	**044**	**172**	**888**

<u>Notice!</u>

- Each step is ***stable***.
- When sorting on digit **`x`**, previous ordering based on digit **`x-1`** is retained.
- This works if buckets are FIFO (First In First Out)
- we remove elements in the order then went in.

LSD

003	**008**	**010**	**212**	**015**	**015**	**020**	**044**	**172**	**888**

Perform binning operation one last time on the final digit.

0	1	2	3	4	5	6	7	8	9

LSD

003	008	010	212	015	015	020	044	172	888

003 008 010 015 015 020 044	172	212						888	
0	1	2	3	4	5	6	7	8	9

LSD

003 008 010 015 015 020 044	172	212						888	
0	1	2	3	4	5	6	7	8	9

Repopulate array with final result:

LSD

003	008	010	015	015	020	044	172	212	888

<u>And we’re done!</u>

- Radix sort is **stable**
- Keys can also be characters (ASCII, radix = 255)
- **MSD** works the same way but we consider digits in reverse order.
- Radix sort is **`O(nw)`**, where **`n`** is the number of keys and **`w`** is the number of digits.
- Thus, Radix sort is better than comparison sorting if **`w < log(n)`**
- As mentioned, the binning operation can be implemented using counting sort where the current digit is the key.
- Counting sort is **`O(n)`**, and we’re doing it for each digit = **`O(nw)`**

How about Most Significant Digit (MSD)?

Radix Sort

How about Most Significant Digit (MSD)?

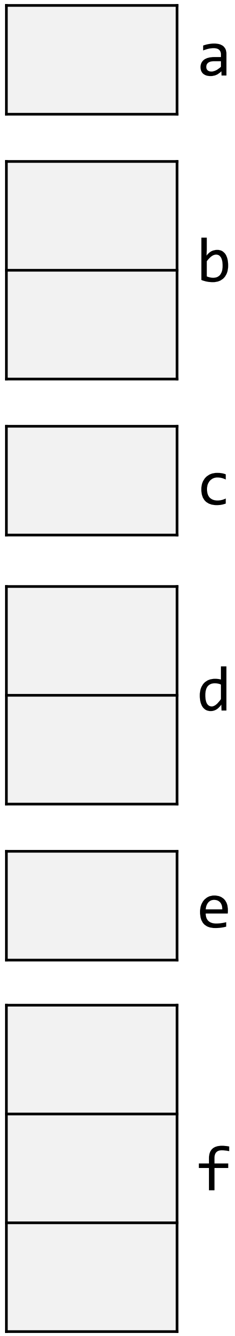
- Most commonly used on strings to achieve ***lexicographic*** (ASCII/Unicode) order.
- Not quite as simple as LSD radix sort – we will recursively sort buckets.
- Let’s try it on an array of strings!

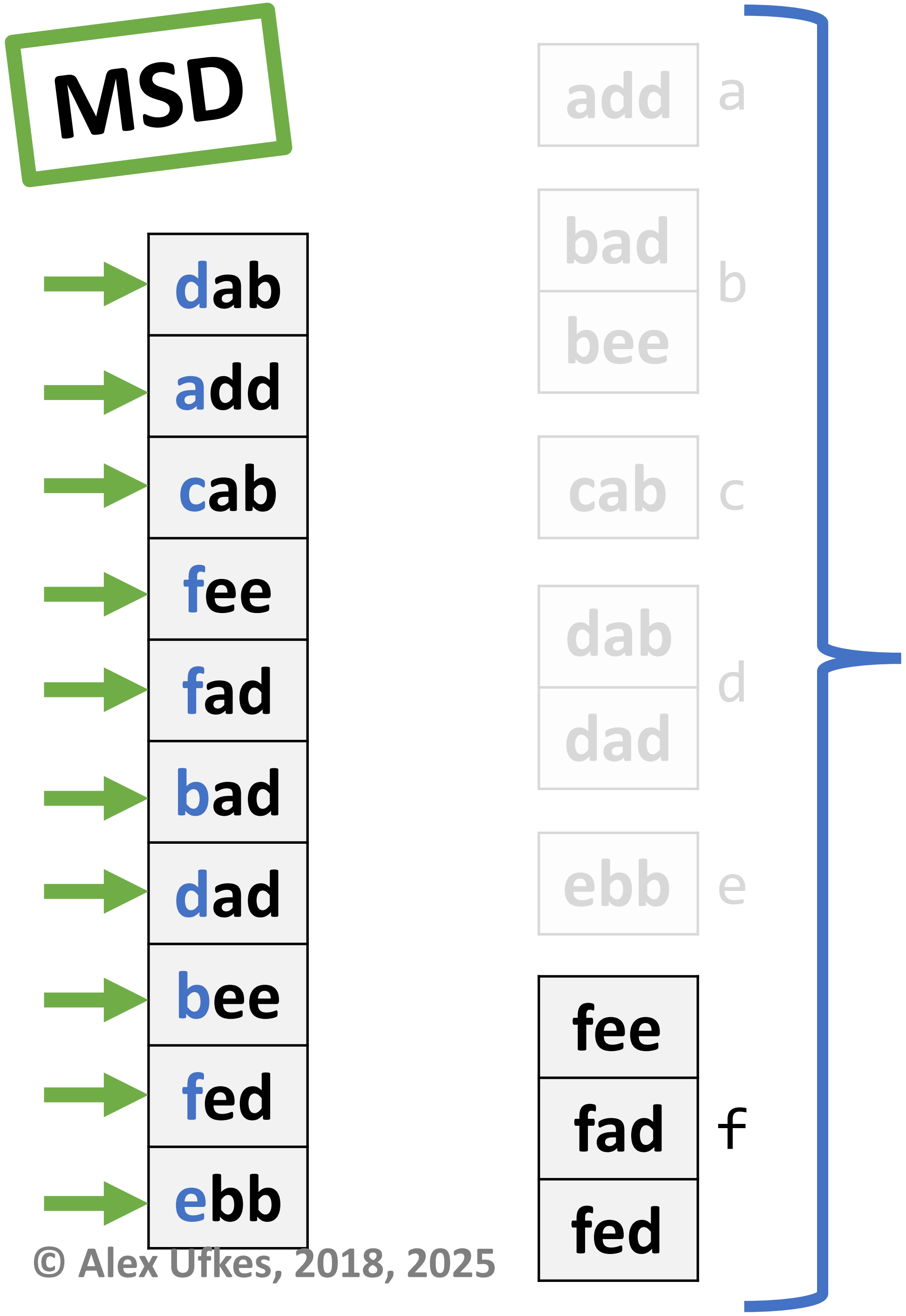
MSD

dab
add
cab
fee
fad
bad
dad
bee
fed
ebb

MSD

dab
add
cab
fee
fad
bad
dad
bee
fed
ebb

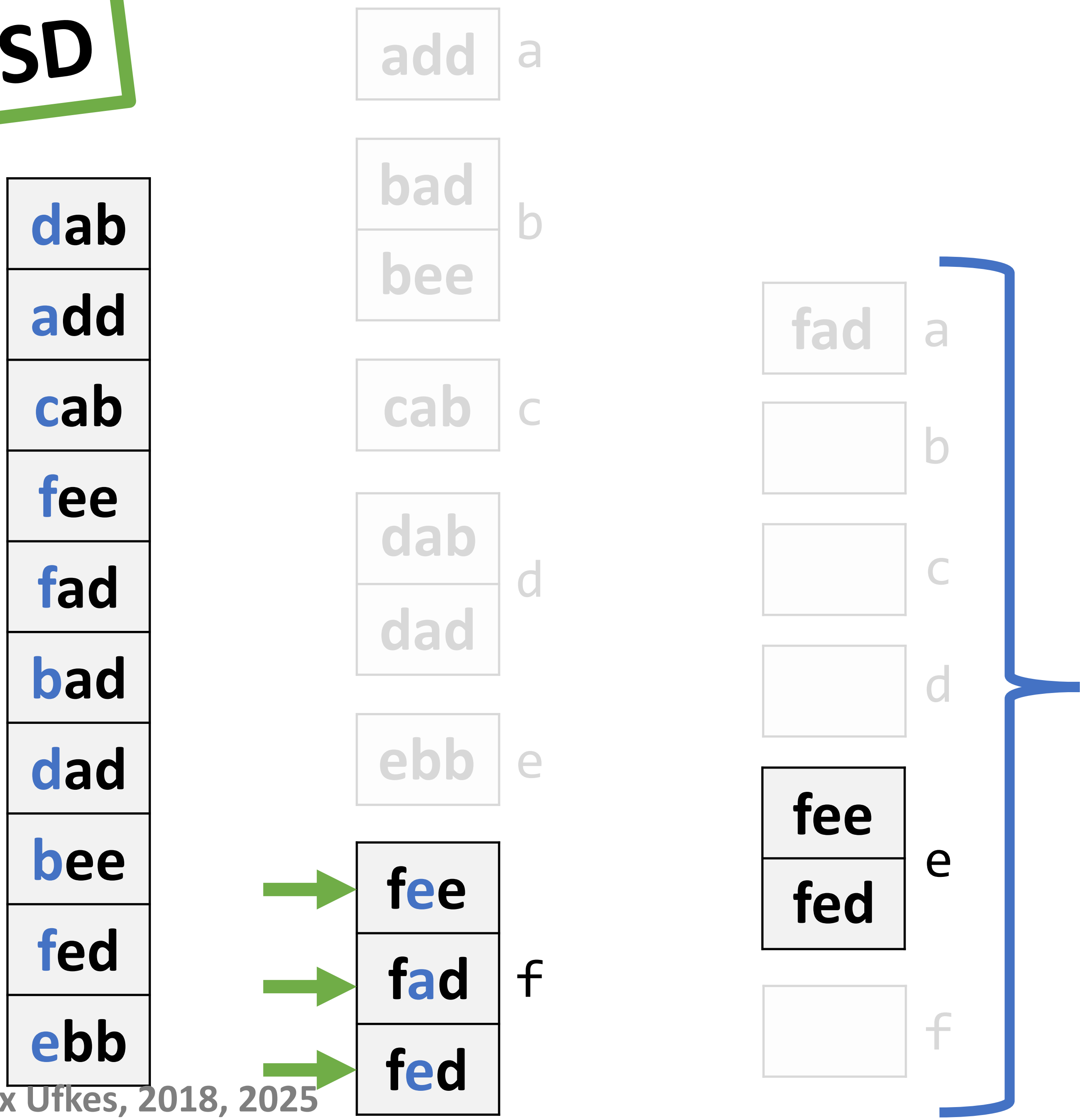




Sort each bin recursively using 2nd letter

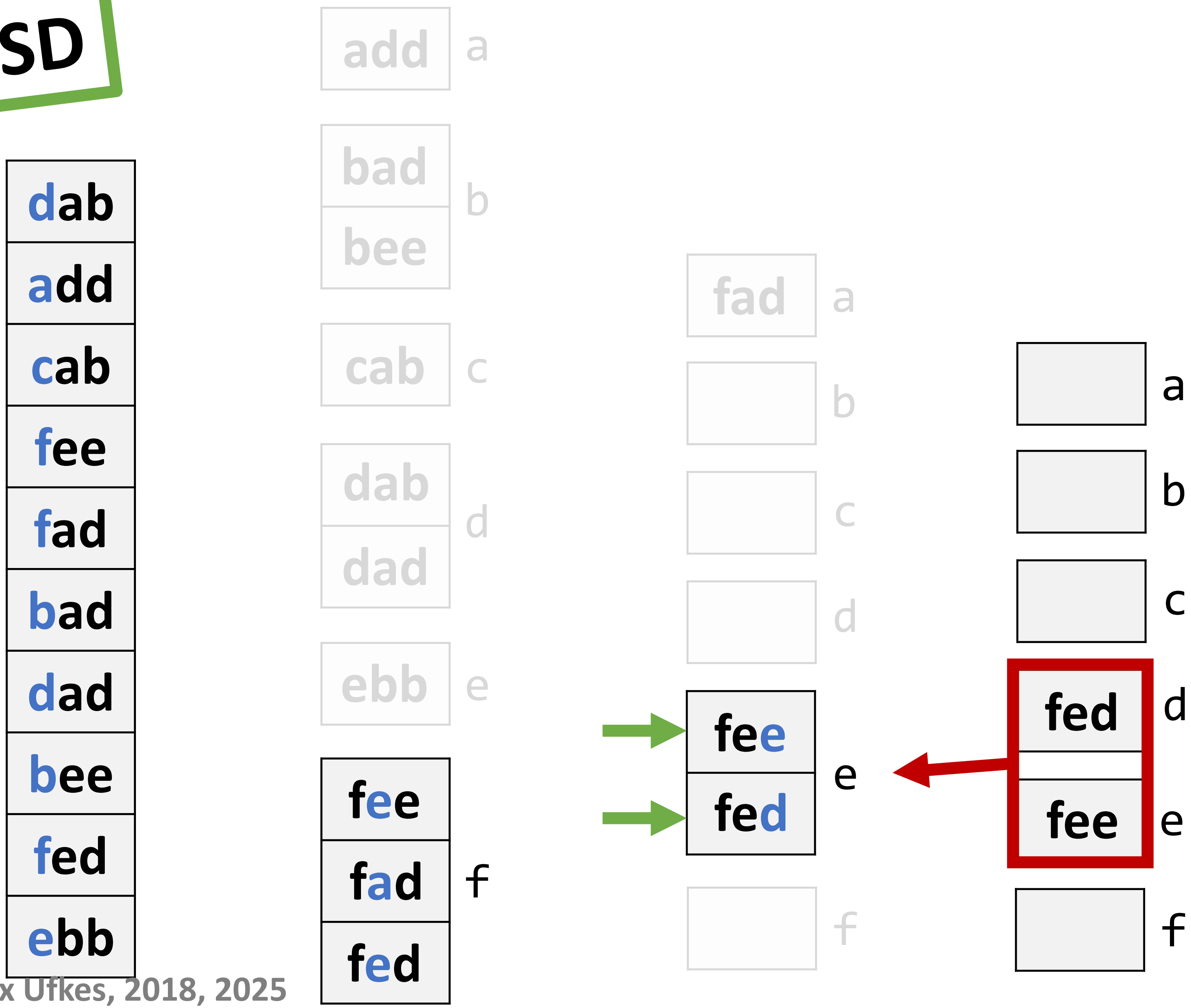
- f is the most interesting bin.
- Let’s see it first

MSD



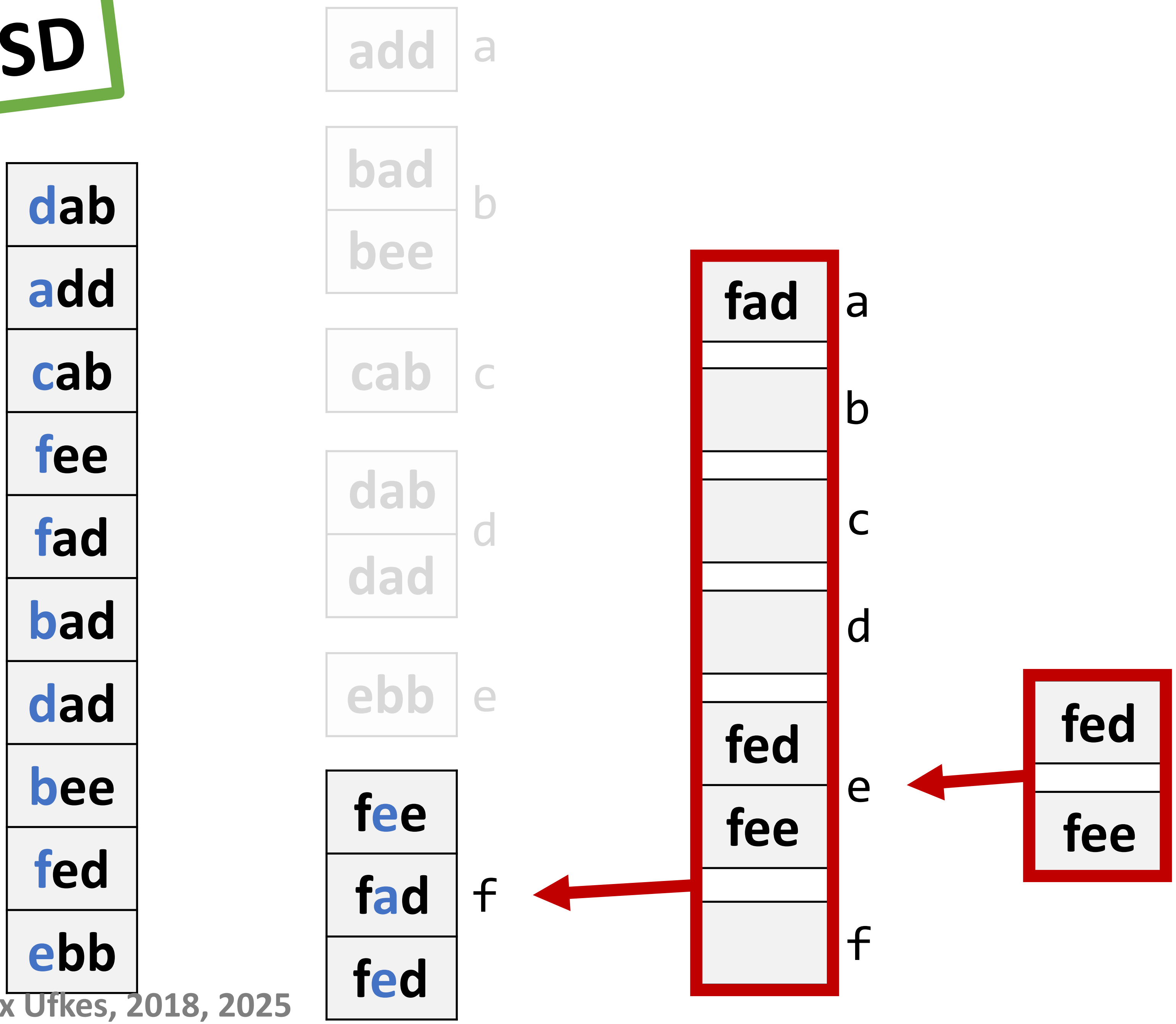
Sort each bin recursively using 3^{rd} letter

MSD



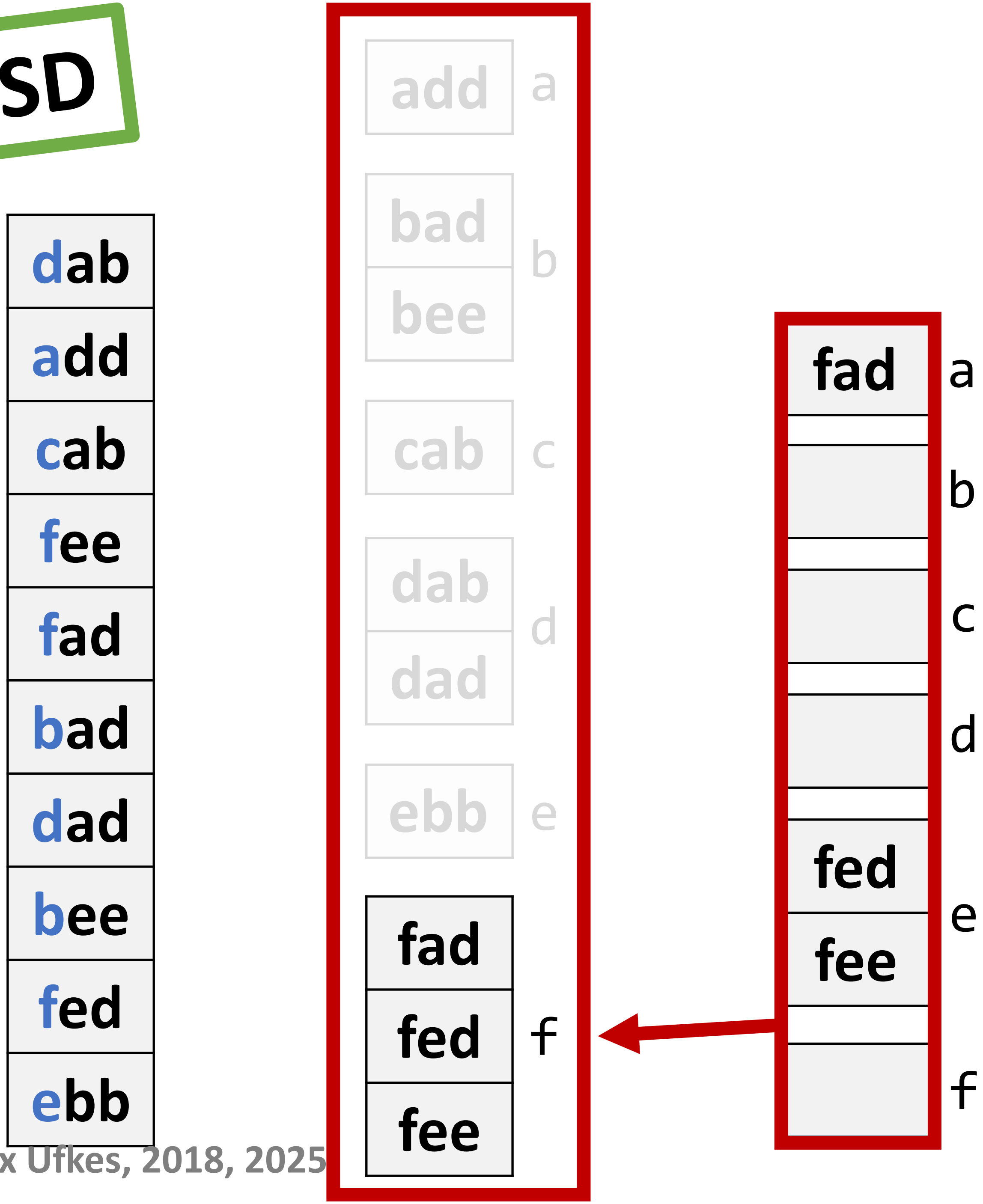
- At this point we hit the base case
- We move back up, concatenating bins as we go

MSD

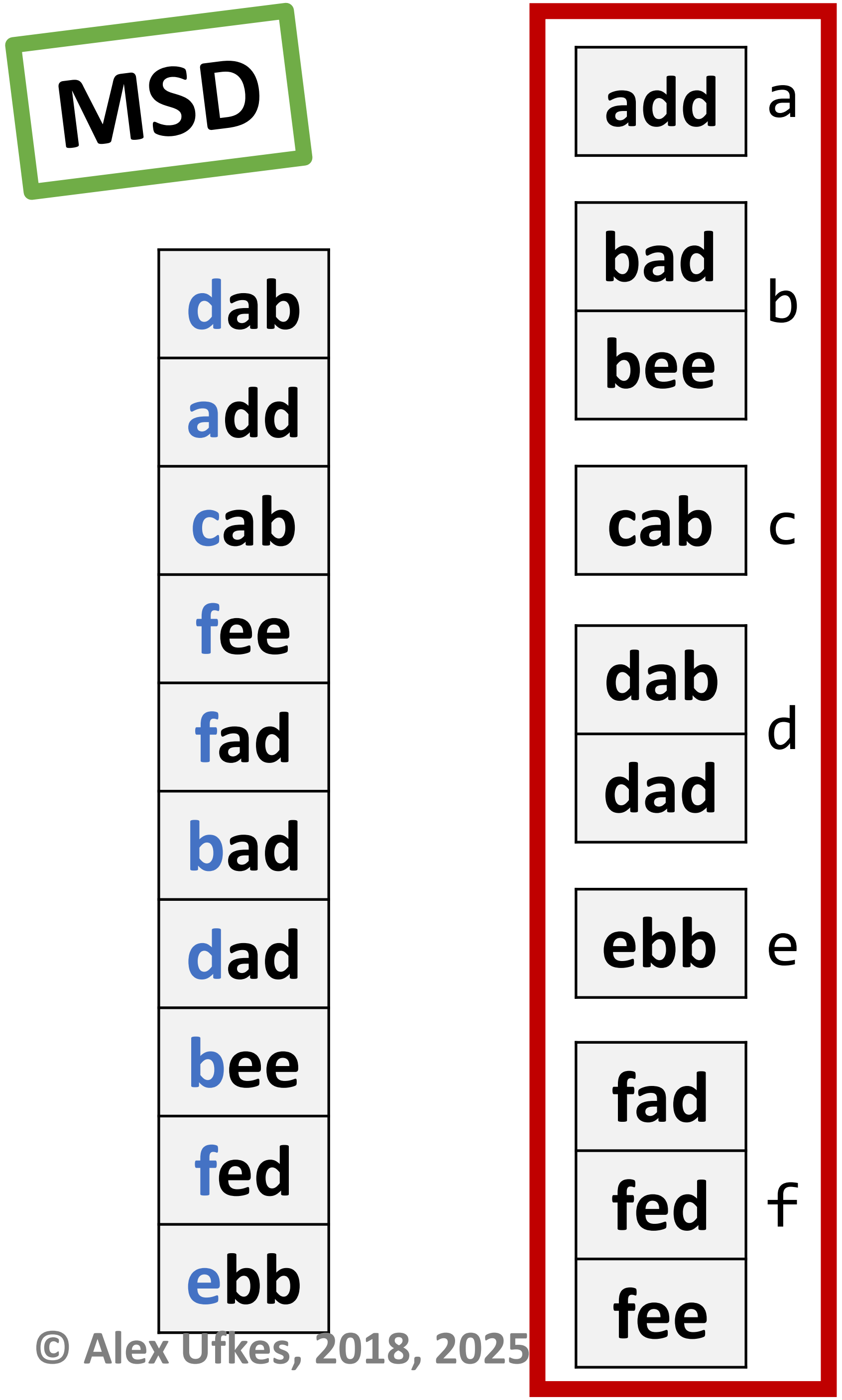


dab
add
cab
fee
fad
bad
dad
bee
fed
ebb
add
a
bad
bee
b
cab
c
dab
dad
d
ebb
e
fee
fad
fed
f
fad
a
b
c
d
fed
fee
e
f
fed
fee

MSD



dab
add
cab
fee
fad
bad
dad
bee
fed
ebb
add
a
bad
bee
b
cab
c
dab
dad
d
ebb
e
fad
fed
fee
f
fad
a
b
c
d
fed
fee
e
f

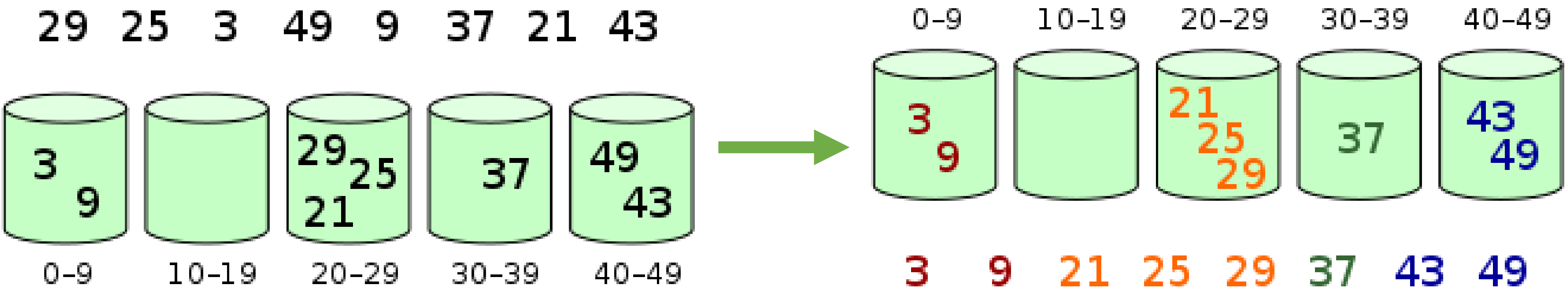


- Assuming we performed the same process on all bins, the final list is now sorted lexicographically.
- MSD Radix Sort is **NOT** guaranteed to be stable!
- **Note:** if the words are different lengths, padding is added on the *right hand* side (left justified).
- It's then up to us to decide if *nothing* should appear before *something*.
- I.e., **a** before **ab**. Typically, it does.

Bucket Sort

Bucket Sort

- Distribute array elements into some number of buckets, then sort each bucket individually.
- Use a traditional sort or recursively apply bucket sort.



Bucket Sort

- Like counting and radix sorts, we want to convert the input array elements into an index.
- This index will determine which bucket we sort that element into.
- Typically, we use some number of significant bits/digits.
- Sorting values from 000 to 999 into 10 bins requires the left-most digit.

000-099	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
0	1	2	3	4	5	6	7	8	9

Bucket Sort

Breaks down when values are not evenly distributed.

2	93	98	95	98	99	99	91	96	96

2									93, 98, 95, 98, 99, 99, 91, 96, 96

- Sorting this bin will dominate the overall running time
- Typically done with insertion sort, $O(n^2)$

000-099	100-199	200-299	300-399	400-499	500-599	600-699	700-799	800-899	900-999
0	1	2	3	4	5	6	7	8	9

Interesting Optimization:

- Rather than sorting bins, copy them, still unsorted, back into the original array, then run insertion sort on original array.
- **Why?** Binning moves every element closer to its final position.
- **Significantly reduces inversions!** Insertion sort's runtime is proportional to the number of inversions.
- Not a guarantee of course, depends on input.

Bucket Sort: Complexity

For bucket sort to be O(n) on average:

- Number of buckets must equal the length of the input array.
- Input array values must be uniformly distributed.
- Clustered data degrades performance.

201	**877**	**004**	**486**	**777**	**919**	**144**	**343**	**555**	**690**

004	**144**	**201**	**343**	**486**	**555**	**690**	**777**	**877**	**919**
0	1	2	3	4	5	6	7	8	9

Binning based on first digit.

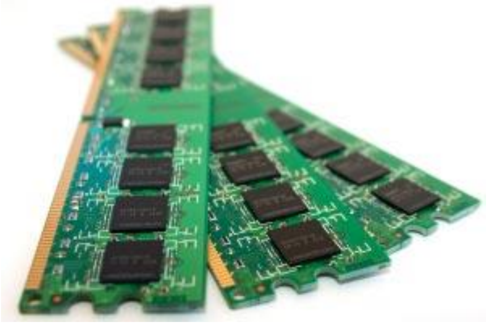
Bucket Sort

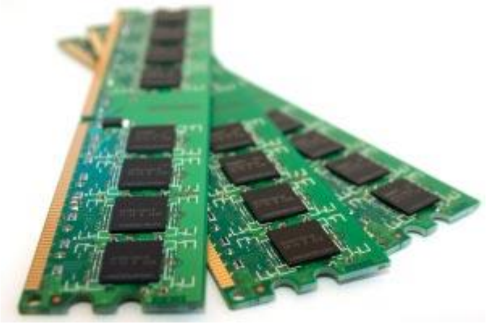
- Notice that bucket sort is not limited to integers!
- We’re not using the keys as indexes
- If the data can be separated into buckets, we can apply bucket sort.

Moving on…

Recall: Bottlenecks

Moving data between different levels of memory is the biggest bottleneck.





Recall: Linked VS Array

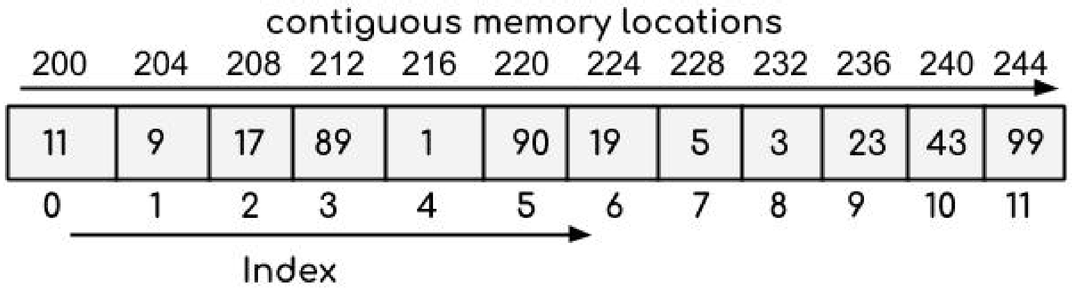
Every Abstract Data Type (ADT) is built on one of two foundations:

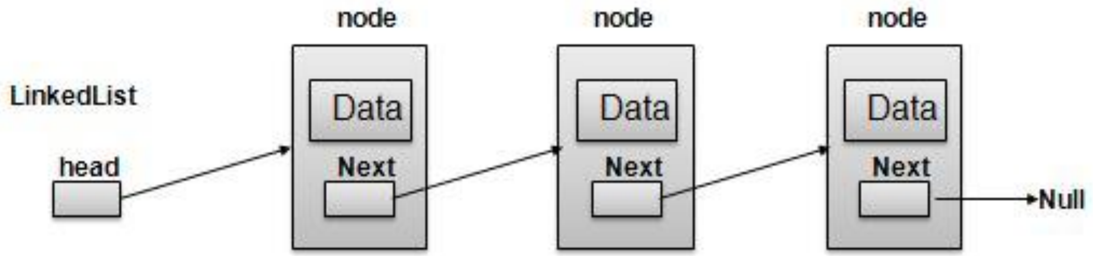
Array:

- Elements are contiguous in memory
- O(1) random access (index)
- O(n) prepend, insertion

Linked data structure:

- Elements, or nodes, are linked via pointers
- Not contiguous in memory!
- O(1) prepend, O(n) indexing





Linked Lists

Linked Lists

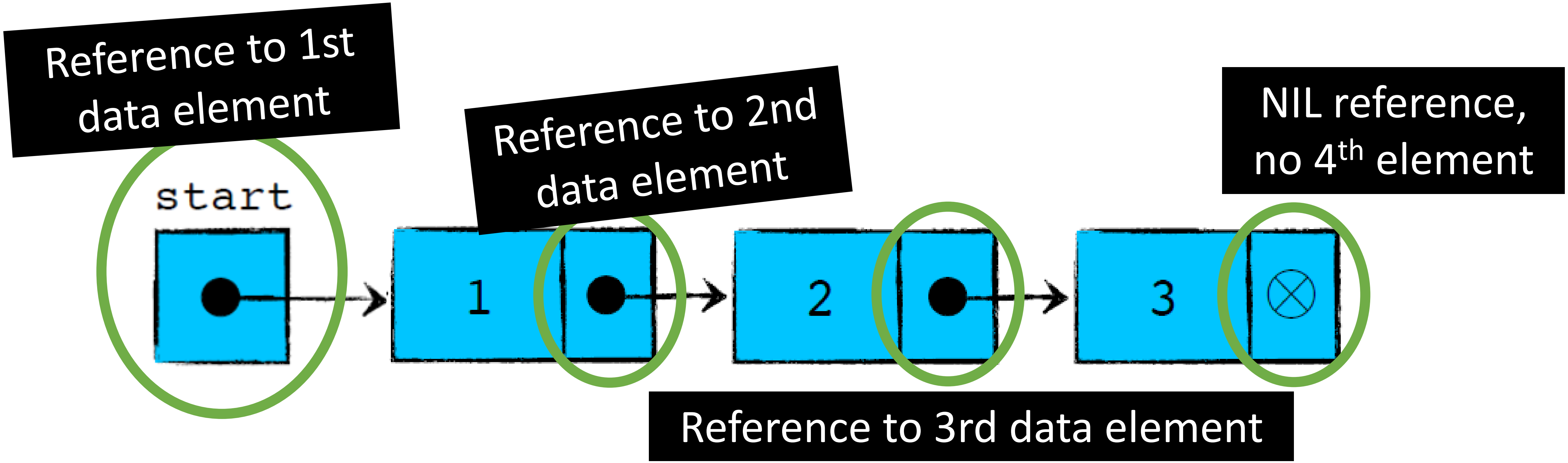
Some object or value

Linked List:

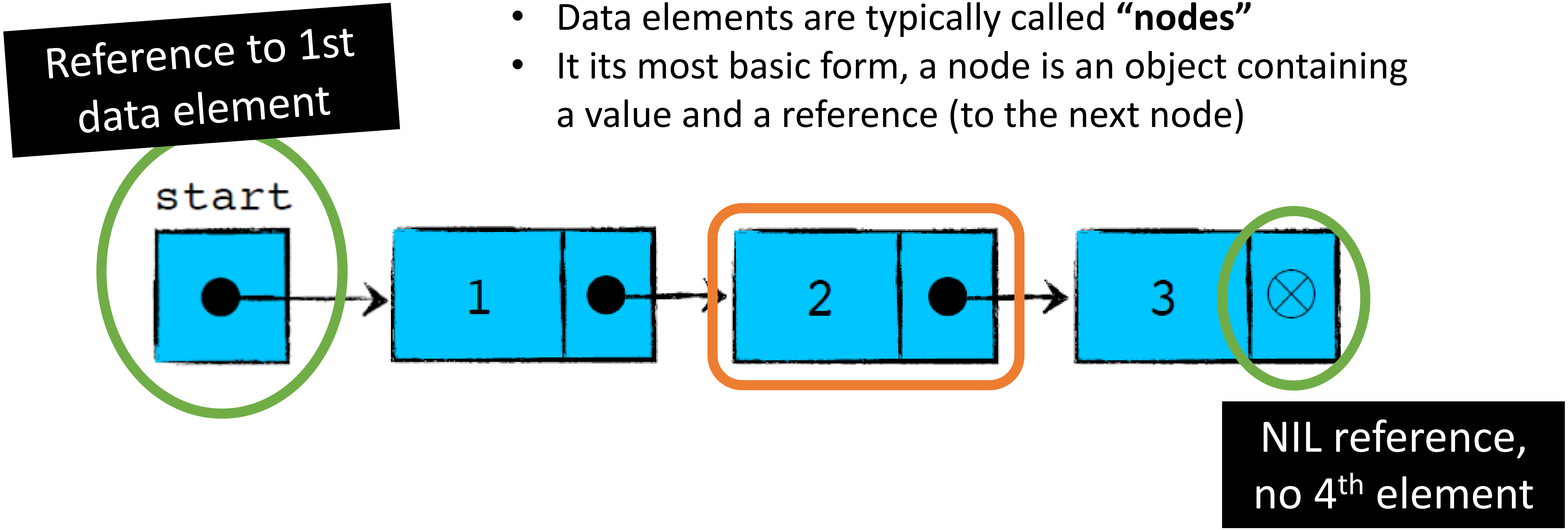
- An ordered set of data elements, each containing a link to its successor (and in some cases, its predecessor).
- Along with arrays, linked lists are used as a base to build other ADTs.

VS. arrays?

- Unlike arrays, linked lists are not contiguous in memory.
- Each element in a linked list contains a reference to the next.
- Fast to add or remove elements, slower to index into.

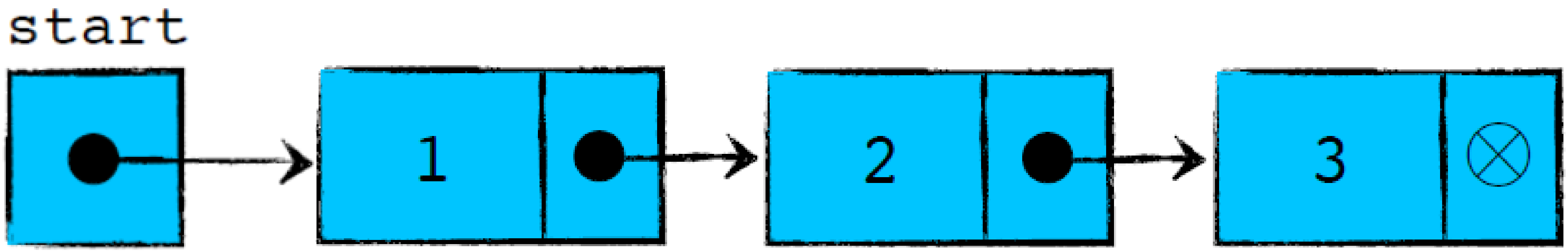

Reference to 1st data element
start
Reference to 2nd data element
1
2
3
NIL reference, no 4th element
Reference to 3rd data element

- Data elements are typically called **“nodes”**
- It its most basic form, a node is an object containing a value and a reference (to the next node)



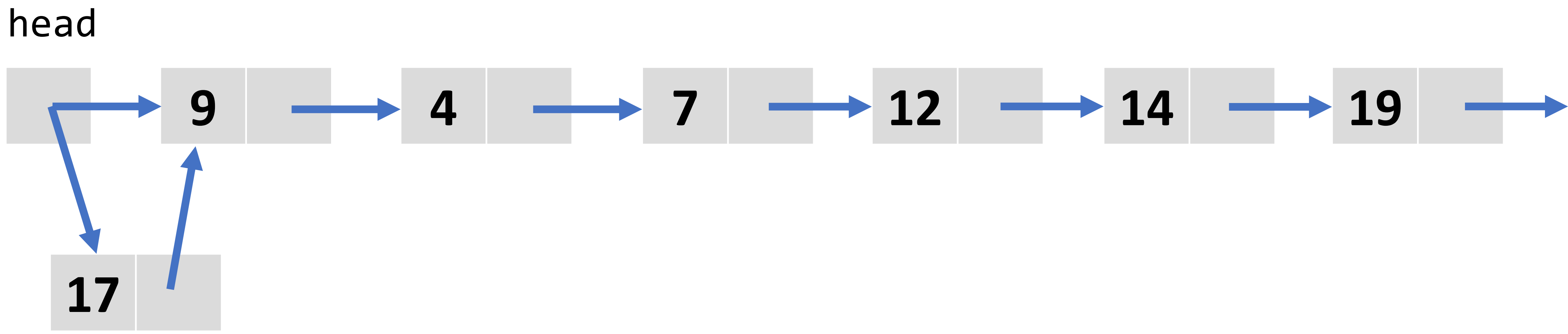
Linked Lists

- To iterate through the list, we simply follow the references beginning with `start` (typically called the *head* of the list)
- Insertion and removal is performed by updating references.
- **Thus:** We need not reallocate or reorganize the entire structure when adding or removing – just update the relevant references.



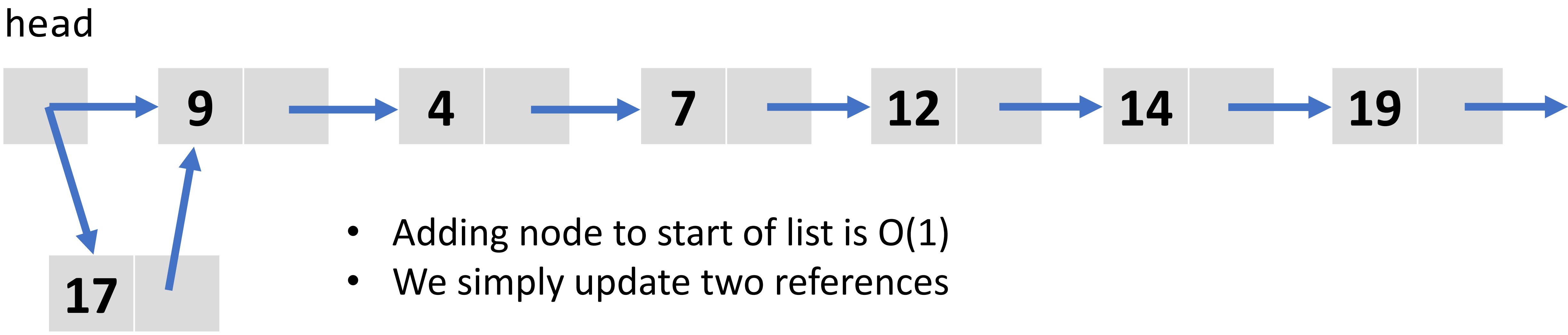
Prepending to a Linked List

Add a new value 17 to the start of the List:



Prepending to a Linked List

Add a new value 17 to the start of the List:



- Adding node to start of list is O(1)
- We simply update two references

VS Arrays

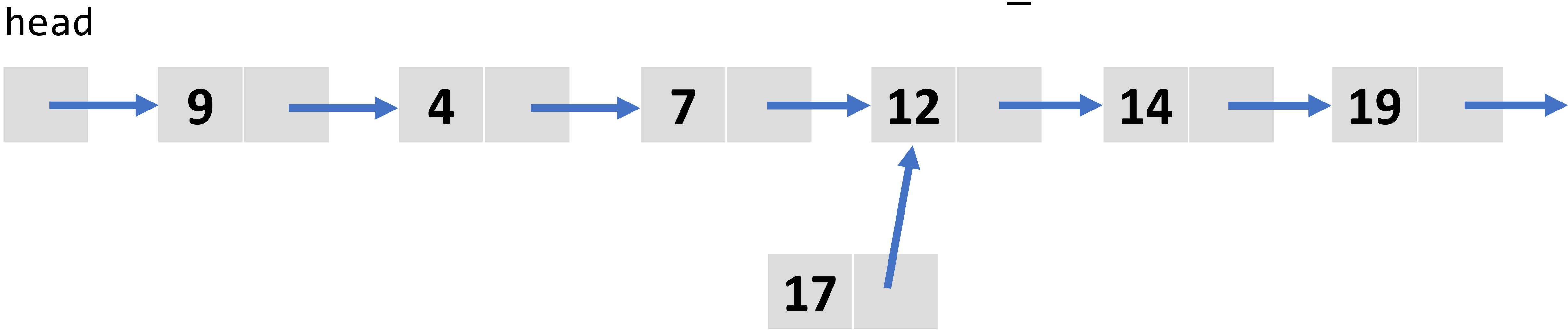
Add a new value 17 to the start of the array:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
a[i]	9	4	7	12	14	19	0	1	2	8	3	7	7

- Shifting elements is bad enough – what if the array has no free space at the end?
- We have to create a new, larger array and copy all the elements over.

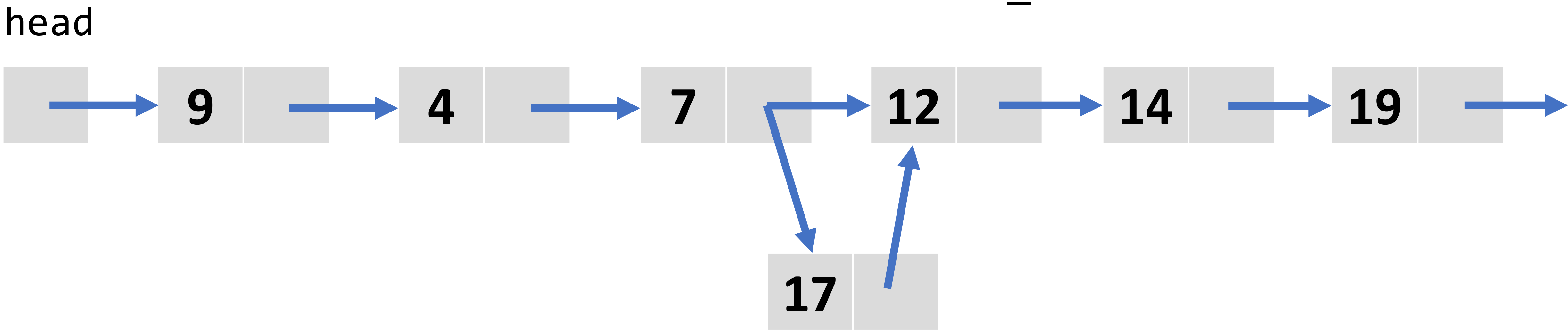
Inserting into a Linked List

Add a new value 17 at index **3**:



Inserting into a Linked List

Add a new value 17 at index **3**:

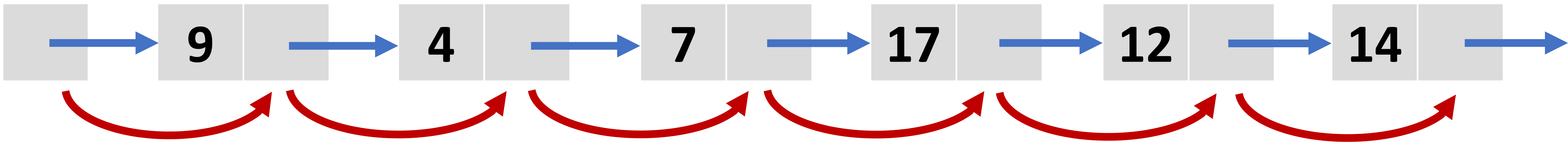


Inserting into a Linked List

We ***only*** have this!

head

Add/Remove arbitrary element?



- Updating references is O(1).
- Thus, adding (or removing) *arbitrary* elements is O(1)….? ***Or is it?***
- Random accesses are O(n). To modify Node at index `i`, we move through `i` references.

VS Arrays?

Arrays are stored *contiguously*.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
a[i]	9	4	7	12	14	19	0	1	2	8	3	7	7

- Accessing array elements involves constant-time address arithmetic.
- Address of `a[i]` = address of `a[0] + i*sizeof(array[0])`

Linked Lists: Advantages

Constant time add/remove…

- …When we don't consider traversal.
- Operating on the start/end of the list is O(1)

More type flexibility

- Nodes can contain anything, as long as they have a reference to the next node.
- Array elements must be homogenous, or indexing doesn't work

Linked Lists: Disadvantages

Requires more memory

- Each element requires an additional reference.

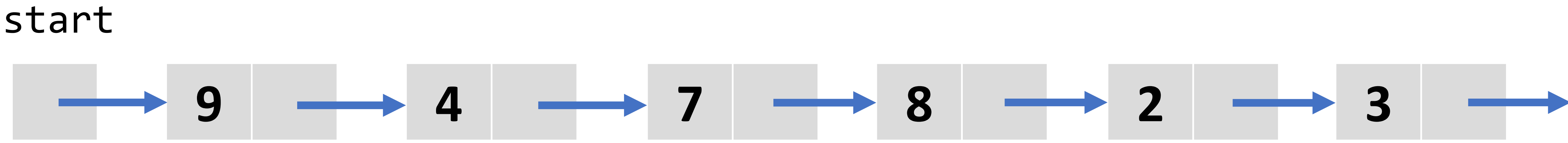
No random access

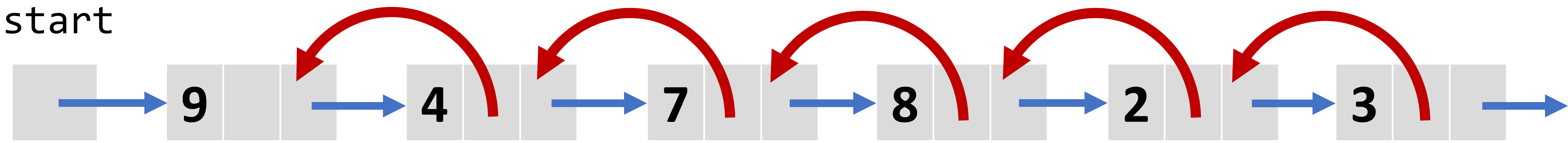
- All accesses start at the beginning (or end).
- We iterate through the list to find arbitrary elements.

As it turns out, there are several ADTs whose operations are defined at their first and/or last elements. Coming up.

Variant: Doubly Linked List

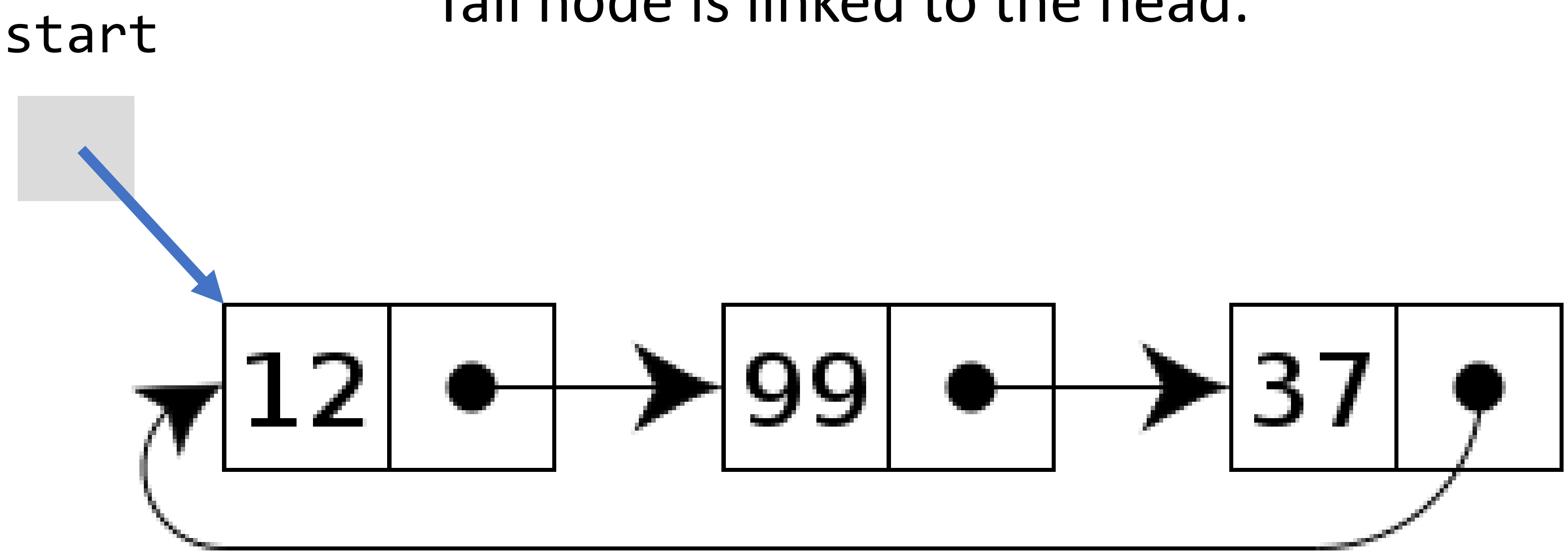
- The linked lists we've seen so far have been *singly* linked.
- This means we can traverse forward through the list, but not backwards. At least not easily.
- The nodes of a ***doubly*** linked list contain two references, one to the *next* node and another to the *previous*.





Variant: Circular

Tail node is linked to the head:

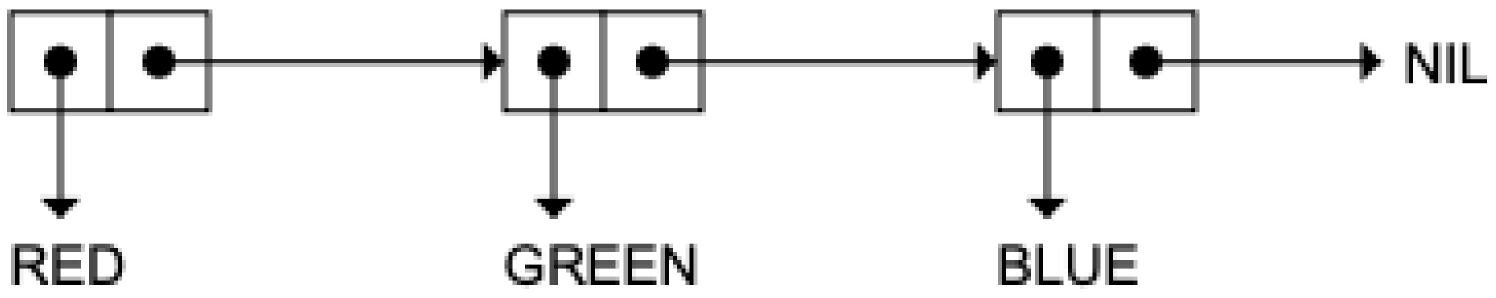


Linked Lists in LISP

Forms are lists, which means LISP code is built from lists!

- Lists in LISP can contain anything – Strings, symbolic atoms, numeric atoms, even other lists.
- Nodes are often called ***cons cells***

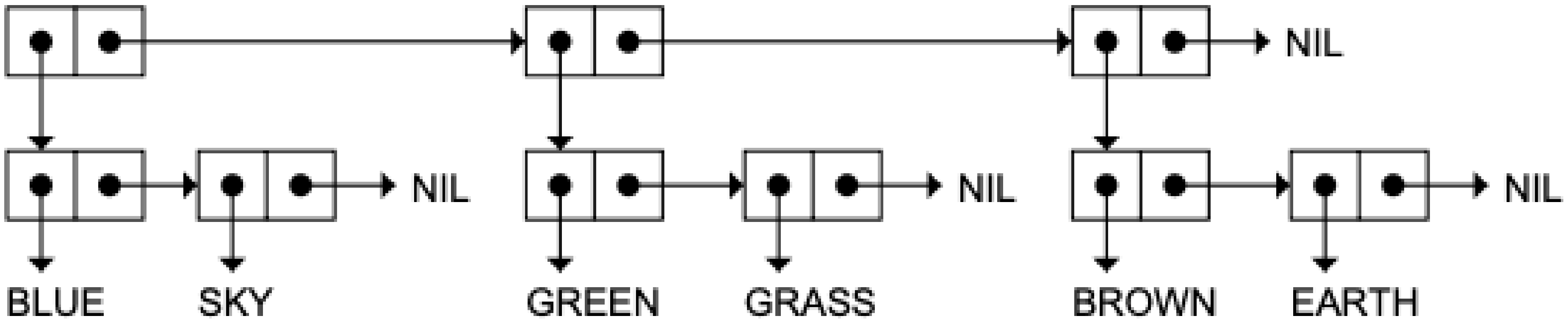
```
(RED GREEN BLUE)
```



Linked Lists in LISP

Lists in LISP can contain anything – Strings, symbolic atoms, numeric atoms, **<u>even other lists</u>**:

`((BLUE SKY) (GREEN GRASS) (BROWN EARTH))`



Linked Lists in LISP

Use the length form to count elements:

```
* (length '(RED GREEN BLUE))
3
* (length '((BLUE SKY) (GREEN GRASS) (BROWN EARTH)))
3
*
```

Don't forget to quote!

- If you don't quote, SBCL will evaluate the list as a form.

Linked Lists in LISP

Use the length form to count elements:

```
* (length '(RED GREEN BLUE))
3
* (length '((BLUE SKY) (GREEN GRASS) (BROWN EARTH)))
3
*
* (length ())
0
* (length NIL)
0
```

Empty List?

- Use **()** or **NIL**

Linked Lists in LISP: Accessing

`FIRST`, `SECOND`, `THIRD`, `REST`

Lisp's primitive functions for extracting elements from a list:

```
(first '(a b c d))    => a
(second '(a b c d))   => b
(third '(a b c d))    => c
```

REST returns a list containing everything ***but*** the first element:

```
(rest '(a b c d))           => (b c d)
(rest (rest '(a b c d)))    => (c d)
```

Linked Lists in LISP: Accessing

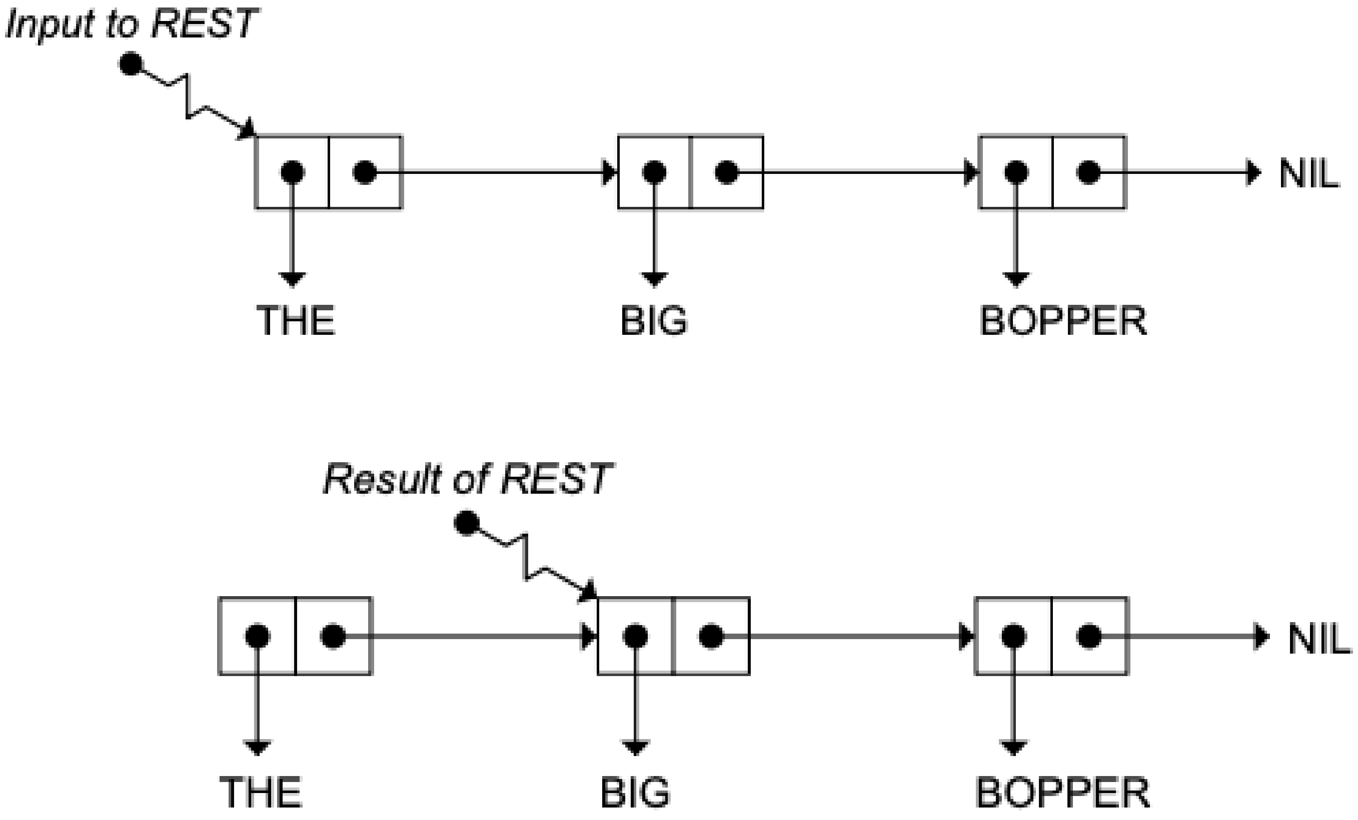
Get the fourth element? Not built in, but we can do it:

```
(defun fourth (a)
  (first (rest (rest (rest a))))
)
```

- When passing a list to a function, only a reference/pointer (to the first cell) is sent.
- We are not sending a copy of the list.

Linked Lists in LISP: Accessing

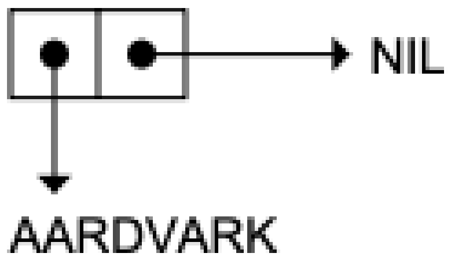
REST also returns a reference:



Linked Lists in LISP: Accessing

Each cons cell points to two things:

1. The element/value for that cell
2. The next cell in the list



- The two halves have obscure names, relics from the early days of computing on an IBM 704
- Left half is called "CAR": Contents of Address portion of Register
- Right half is called "CDR": Contents of Decrement portion of Register
- CAR and CDR are also names of built-in lisp functions

Linked Lists in LISP: CAR and CDR

CAR is the same as FIRST: **`(car '(a b c d)) => a`**
CDR is the same as REST: **`(cdr '(a b c d)) => (b c d)`**

Other built-in combinations:

```
(caar '((a b c) (d e f)))  => a
(cadr '((a b c) (d e f)))  => (d e f)
(cddar '((a b c) (d e f))) => (c)
```

Read **dda** portion right to left:

- **a**, then **d**, then **d**
- First, then rest, then rest.

The CONStructor

CONS creates a cons cell:

- takes two arguments – an element, and a list
- Returns a pointer to a new cons cell whose CAR points to the first parameter and whose CDR points to the second

```
* (cons 'sink '(or swim))
(SINK OR SWIM)
* (cons 'sink ())
(SINK)
* (cons '(or swim) ())
((OR SWIM))
* (cons 'bond '(james bond))
(BOND JAMES BOND)
```

The CONStructor: Recursive

Length of list

Value of each element

Running list, initially NIL

```
(defun mymake-list-rec (n element &optional (acc nil))
  (if (= n 0) acc ; If n is zero, we're done (return acc)
    (mymake-list-rec (1- n) element (cons element acc))
  )
)
```

Recursive call:

- n-1 elements remaining to add
- Running result updated with **cons** call

The CONStructor: Iterative

```
(defun mymake-list-it (n elem)
  (let ((acc nil))
  (dotimes (i n acc)
    (setf acc (cons elem acc)))
  )
)
```

Initialize acc to NIL

Iterate n times, evaluate to acc

Update acc each iteration

quicksort.lisp | make-list.lisp

make-list.lisp > ...

```lisp
(defun mymake-list-rec (n element &optional (acc nil))
    (if (= n 0) acc ; If n is zero, we’re done. Return acc.
        (mymake-list-rec (1- n) element (cons element acc))
    )
)

(defun mymake-list-it (n elem)
    (let ((acc nil))
    (dotimes (i n acc)
        (setf acc (cons elem acc)))
    )
)
```

```
* (load "make-list.lisp")
T
* (mymake-list-rec 3 'a)
(A A A)
* (mymake-list-it 7 'b)
(B B B B B B B)
```

Built-in Constructors

QUOTE, MAKE-LIST, LIST

```
> '("hello" world 111) ; This creates a literal (constant) list.
("hello" WORLD 111)    ; Its contents should not be changed.
> (make-list 3)
(NIL NIL NIL)
> (make-list 3 :initial-element 'a)
(A A A)
> (make-list 3 :initial-contents '(a b c))
(A B C)
> (list "hello" 'world 111)
("hello" WORLD 111)
```

Built-in Traversal: `DOLIST`

```
(DOLIST (var list-form [result-form]) body-form*)
```

First: list-form is evaluated once to produce a list.
Then: The body-form* is evaluated once for each item in the list with the variable var holding the value of the item.
Lastly: If result-form is provided, it is evaluated, and its value is returned; otherwise DOLIST returns NIL

Built-in Traversal: DOLIST

```
* (dolist (x '(1 2 3)) (print x))
1
2
3
NIL
* (dolist (x '(1 2 3)) (print x) (if (evenp x) (return "HI")))
1
2
"HI"
```

Can exit DOLIST early with return, just like other loops

Linked Lists and Arrays provide a foundation for implementing more complex ADTs

Next class we'll implement our own List type from scratch in LISP

We'll then build other ADTs using lists

In Summary

- Integer sorting, Radix & Counting sort
- Linked lists in LISP, cons cell structure