Data Structures and Algorithms

CS245-2015S-12

Non-Comparison Sorts

David Galles

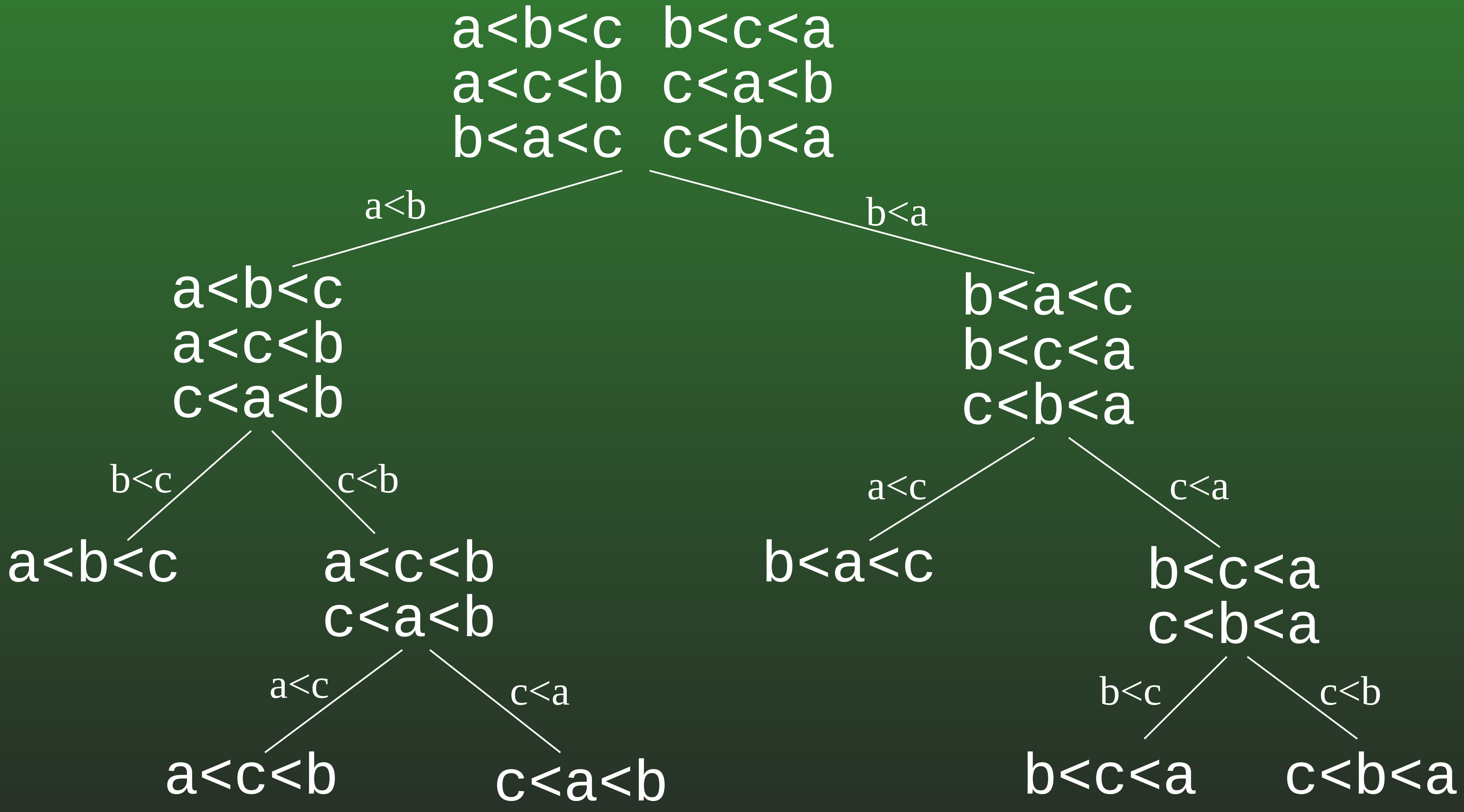
Department of Computer Science
University of San Francisco

Comparison Sorting

- Comparison sorts work by comparing elements
  - Can only compare 2 elements at a time
  - Check for $<$, $>$, $=$.
- All the sorts we have seen so far (Insertion, Quick, Merge, Heap, etc.) are comparison sorts
- If we know nothing about the list to be sorted, we need to use a comparison sort

12-1: Decision Trees

Insertion Sort on list $\{a, b, c\}$



a<b<c b<c<a
a<c<b c<a<b
b<a<c c<b<a

a<b

b<a

a<b<c
a<c<b
c<a<b

b<a<c
b<c<a
c<b<a

b<c

c<b

a<c

c<a

a<b<c

a<c<b
c<a<b

b<a<c

b<c<a
c<b<a

a<c

c<a

b<c

c<b

a<c<b

c<a<b

b<c<a

c<b<a

12-2: Decision Trees

- Every comparison sorting algorithm has a decision tree
- What is the best-case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?

12-3: Decision Trees

- Every comparison sorting algorithm has a decision tree
- What is the best-case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?
  - (The depth of the shallowest leaf) + 1
- What is the worst case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?

12-4: **Decision Trees**

- Every comparison sorting algorithm has a decision tree
- What is the best-case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?
  - (The depth of the shallowest leaf) + 1
- What is the worst case number of comparisons for a comparison sorting algorithm, given the decision tree for the algorithm?
  - The height of the tree – (depth of the deepest leaf) + 1

12-5: Decision Trees

- What is the largest number of nodes for a tree of depth d?

12-6: **Decision Trees**

- What is the largest number of nodes for a tree of depth d?
  - 2^d
- What is the minimum height, for a tree that has n leaves?

12-7: Decision Trees

- What is the largest number of nodes for a tree of depth d?
  - 2^d
- What is the minimum height, for a tree that has n leaves?
  - $\lg n$
- How many leaves are there in a decision tree for sorting n elements?

12-8: Decision Trees

- What is the largest number of nodes for a tree of depth d?
  - 2^d
- What is the minimum height, for a tree that has n leaves?
  - $\lg n$
- How many leaves are there in a decision tree for sorting n elements?
  - $n!$
- What is the minimum height, for a decision tree for sorting n elements?

12-9: Decision Trees

- What is the largest number of nodes for a tree of depth d?
    - 2^d
- What is the minimum height, for a tree that has n leaves?
    - $\lg n$
- How many leaves are there in a decision tree for sorting n elements?
    - $n!$
- What is the minimum height, for a decision tree for sorting n elements?
    - $\lg n!$

12-10: $\lg(n!) \in \Omega(n \lg n)$

$$\begin{aligned}
\lg(n!) &= \lg(n * (n-1) * (n-2) * \ldots * 2 * 1) \\
&= (\lg n) + (\lg(n-1)) + (\lg(n-2)) + \ldots \\
&\quad +(\lg 2) + (\lg 1) \\
&\geq \underbrace{(\lg n) + (\lg(n-1)) + \ldots + (\lg(n/2))}_{n/2 \text{ terms}} \\
&\geq \underbrace{(\lg n/2) + (\lg(n/2)) + \ldots + \lg(n/2)}_{n/2 \text{ terms}} \\
&= (n/2)\lg(n/2) \\
&\in \Omega(n \lg n)
\end{aligned}$$

Sorting Lower Bound

- All comparison sorting algorithms can be represented by a decision tree with $n!$ leaves
- Worst-case number of comparisons required by a sorting algorithm represented by a decision tree is the height of the tree
- A decision tree with $n!$ leaves must have a height of at least $n \lg n$
- All comparison sorting algorithms have worst-case running time $\Omega(n \lg n)$

Counting Sort

- Sorting a list of n integers
- We know all integers are in the range $0 \ldots m$
- We can potentially sort the integers faster than $n \lg n$
- Keep track of a “Counter Array” C:
  - $C[i]$ = # of times value i appears in the list

Example: 3 1 3 5 2 1 6 7 8 1

1	2	3	4	5	6	7	8	9

Counting Sort Example

`3135216781`

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

Counting Sort Example

135216781

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

Counting Sort Example

35216781

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

Counting Sort Example

`5216781`

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

Counting Sort Example

`216781`

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

Counting Sort Example

`16781`

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

Counting Sort Example

`6781`

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

Counting Sort Example

781

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

Counting Sort Example

81

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

Counting Sort Example

1

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

Counting Sort Example

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

12-24: Counting Sort Example

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

1 1 1 2 3 3 5 6 7 8

$\Theta()$ of Counting Sort

- What its the running time of Counting Sort?
- If the list has n elements, all of which are in the range $0 \ldots m$:

$\Theta()$ of Counting Sort

- What its the running time of Counting Sort?
- If the list has n elements, all of which are in the range $0 \ldots m$:
    - Running time is $\Theta(n + m)$
- What about the $\Omega(n \lg n)$ bound for all sorting algorithms?

$\Theta()$ of Counting Sort

- What its the running time of Counting Sort?
- If the list has n elements, all of which are in the range $0 \ldots m$:
  - Running time is $\Theta(n + m)$
- What about the $\Omega(n \lg n)$ bound for all sorting algorithms?
  - For *Comparison Sorts*, which allow for sorting arbitrary data. What happens when m is very large?

Binsort

- Counting Sort will need some modification to allow us to sort *records* with integer keys, instead of just integers.
- Binsort is much like Counting Sort, except that in each index i of the counting array C:
  - Instead of storing the *number* of elements with the value i, we store a *list* of all elements with the value i.

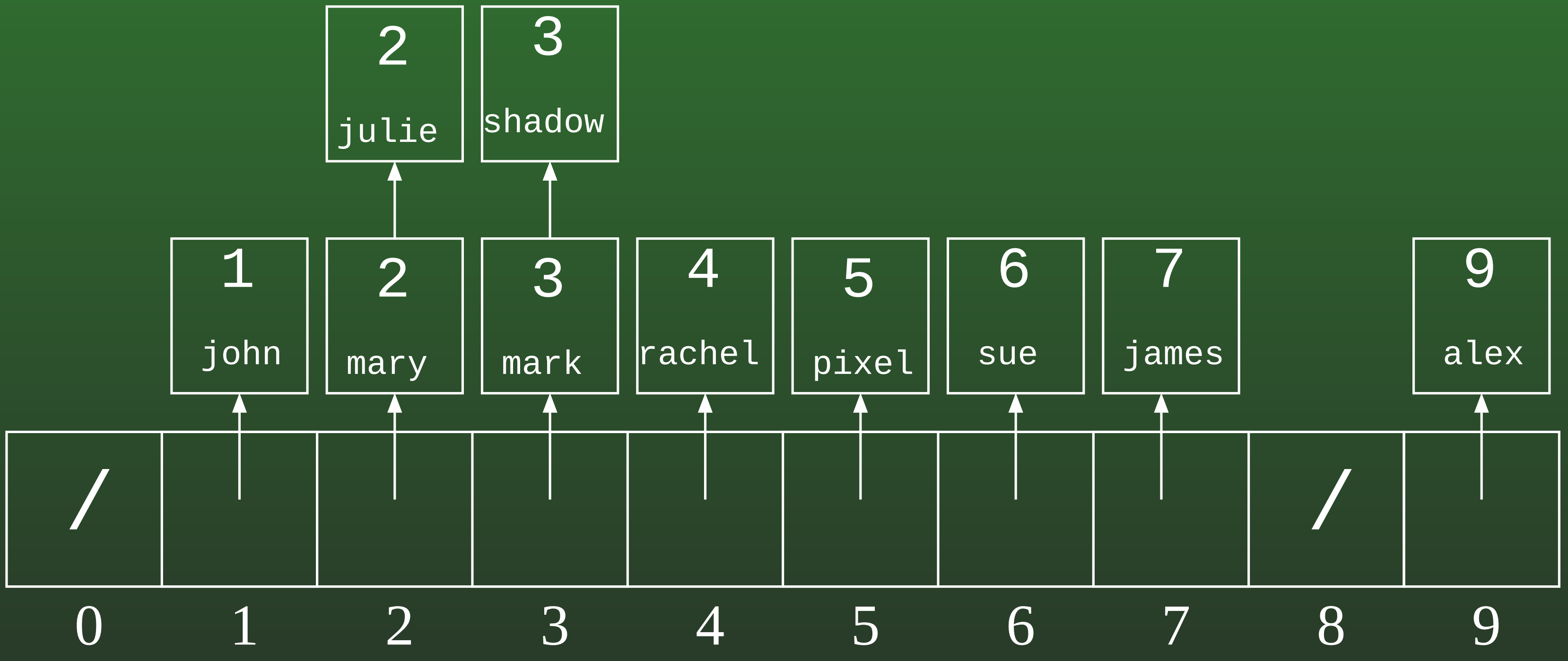
12-29: Binsort Example

3	1	2	6	2	4	5	3	9	7	key
`mark`	`john`	`mary`	`sue`	`julie`	`rachel`	`pixel`	`shadow`	`alex`	`james`	data

/	/	/	/	/	/	/	/	/	/
0	1	2	3	4	5	6	7	8	9

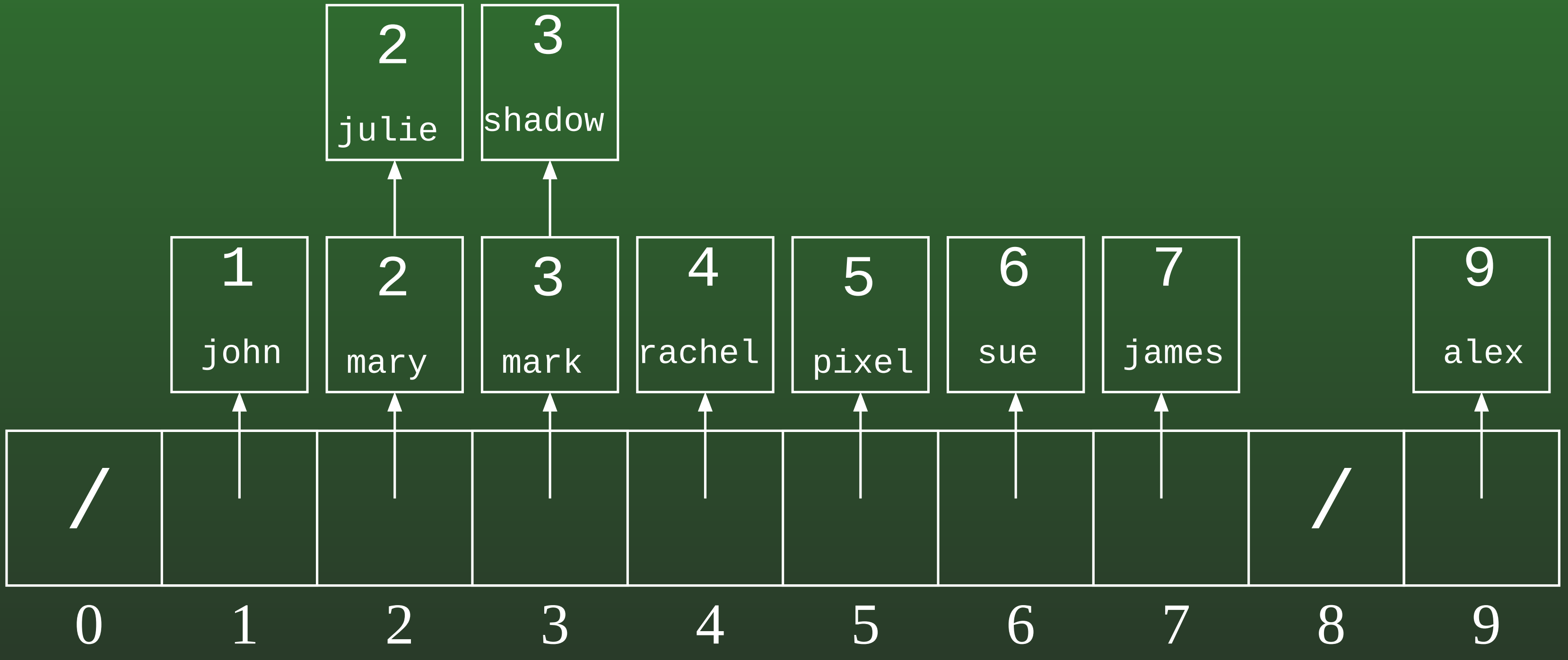
12-30: Binsort Example

3	1	2	6	2	4	5	3	9	7	key
mark	john	mary	sue	julie	rachel	pixel	shadow	alex	james	data



Binsort Example

1	2	2	3	3	4	5	6	7	9	key
john	mary	julie	mark	shadow	rachel	pixel	sue	james	alex	data



12-32: Bucket Sort

- Expand the “bins” in Bin Sort to “buckets”
- Each bucket holds a range of key values, instead of a single key value
- Elements in each bucket are sorted.

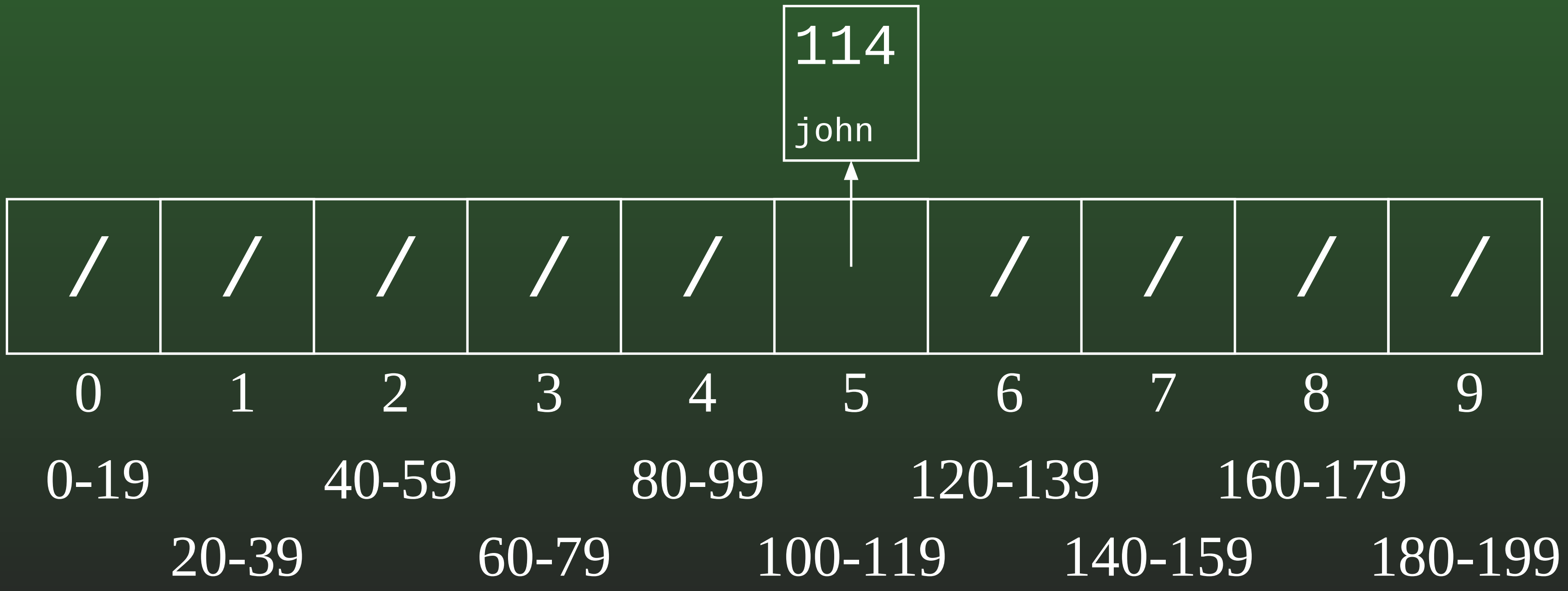
12-33: Bucket Sort Example

114	26	50	180	44	111	4	95	196	170	key
john	mary	julie	mark	shadow	rachel	pixel	sue	james	alex	data

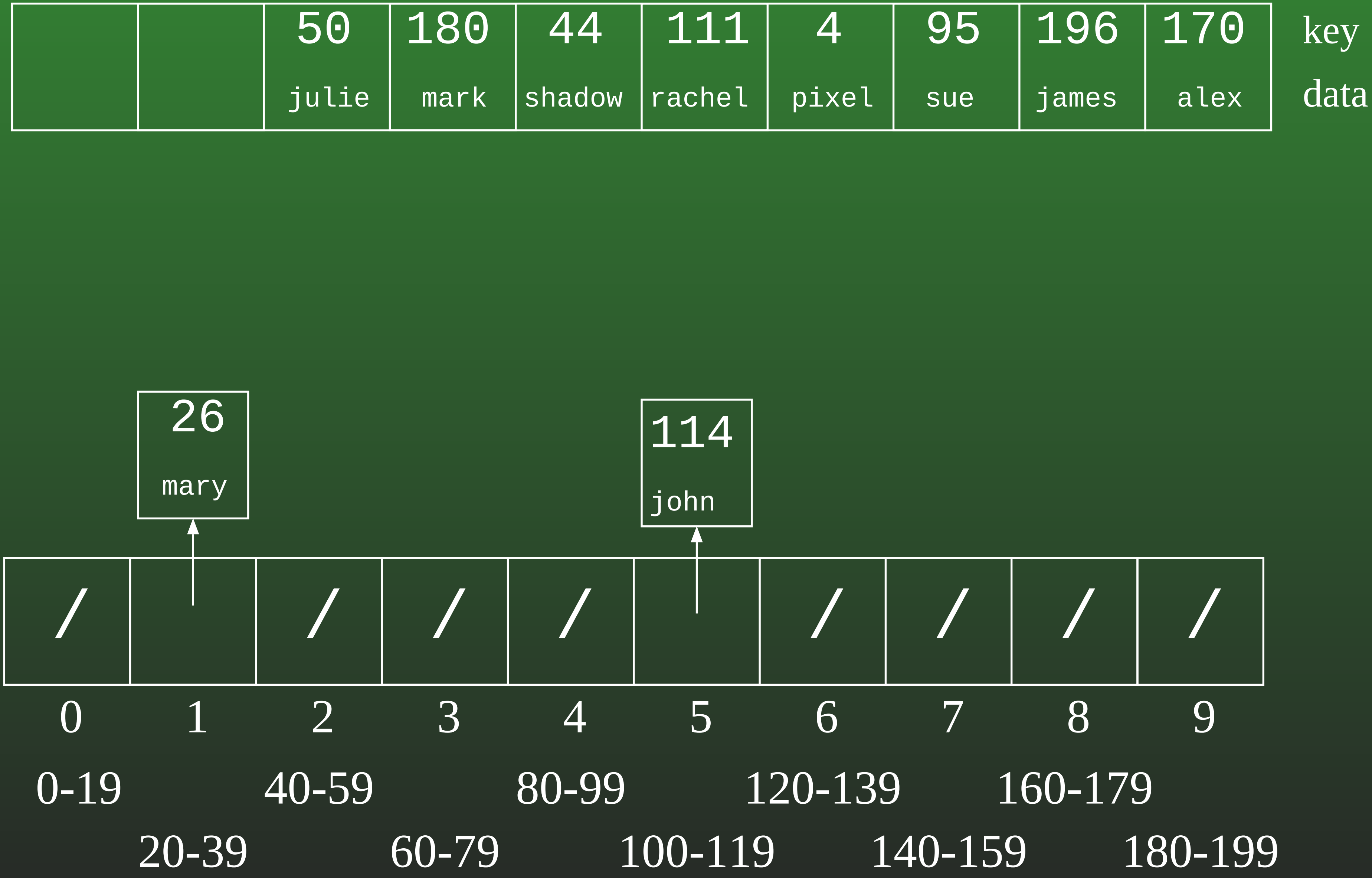
/	/	/	/	/	/	/	/	/	/
0	1	2	3	4	5	6	7	8	9
0-19	20-39	40-59	60-79	80-99	100-119	120-139	140-159	160-179	180-199

Bucket Sort Example

	26	50	180	44	111	4	95	196	170	key
	mary	julie	mark	shadow	rachel	pixel	sue	james	alex	data



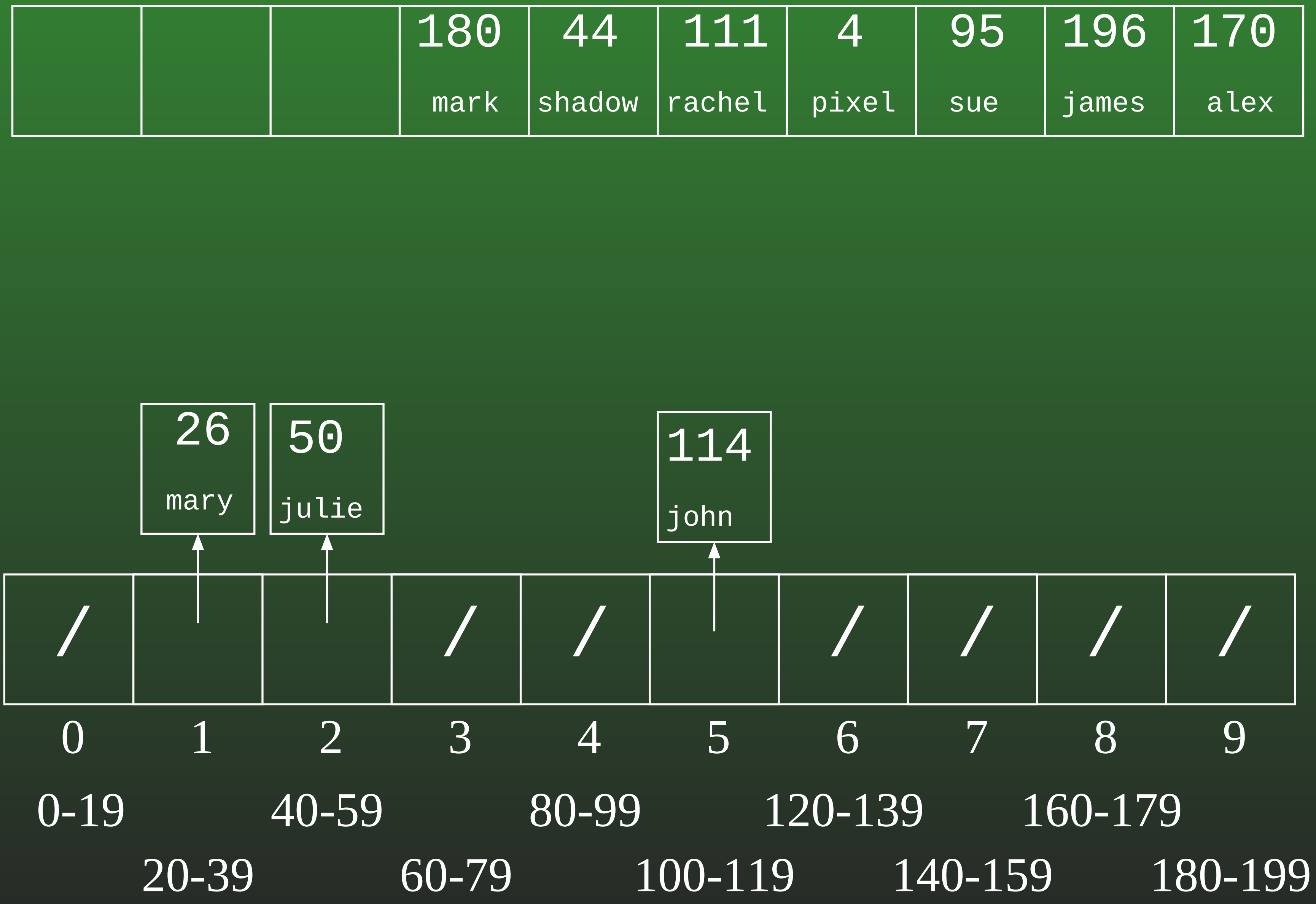
12-35: Bucket Sort Example



		50	180	44	111	4	95	196	170	key
		julie	mark	shadow	rachel	pixel	sue	james	alex	data

/	26 mary	/	/	/	114 john	/	/	/	/
0	1	2	3	4	5	6	7	8	9
0-19	20-39	40-59	60-79	80-99	100-119	120-139	140-159	160-179	180-199

Bucket Sort Example



			180	44	111	4	95	196	170	key
			mark	shadow	rachel	pixel	sue	james	alex	data

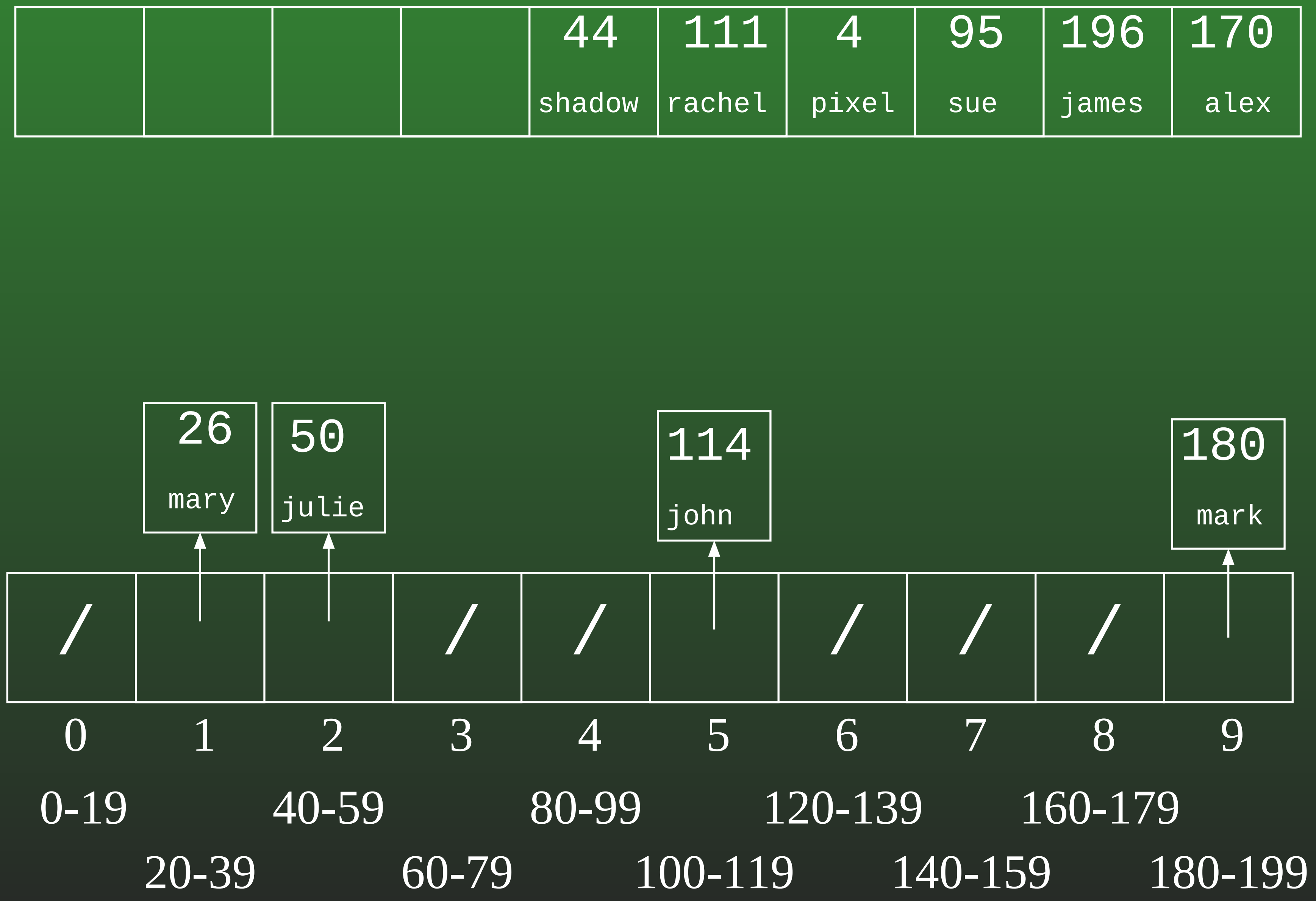
26 mary

50 julie

114 john

/			/	/		/	/	/	/
0	1	2	3	4	5	6	7	8	9
0-19	20-39	40-59	60-79	80-99	100-119	120-139	140-159	160-179	180-199

12-37: Bucket Sort Example

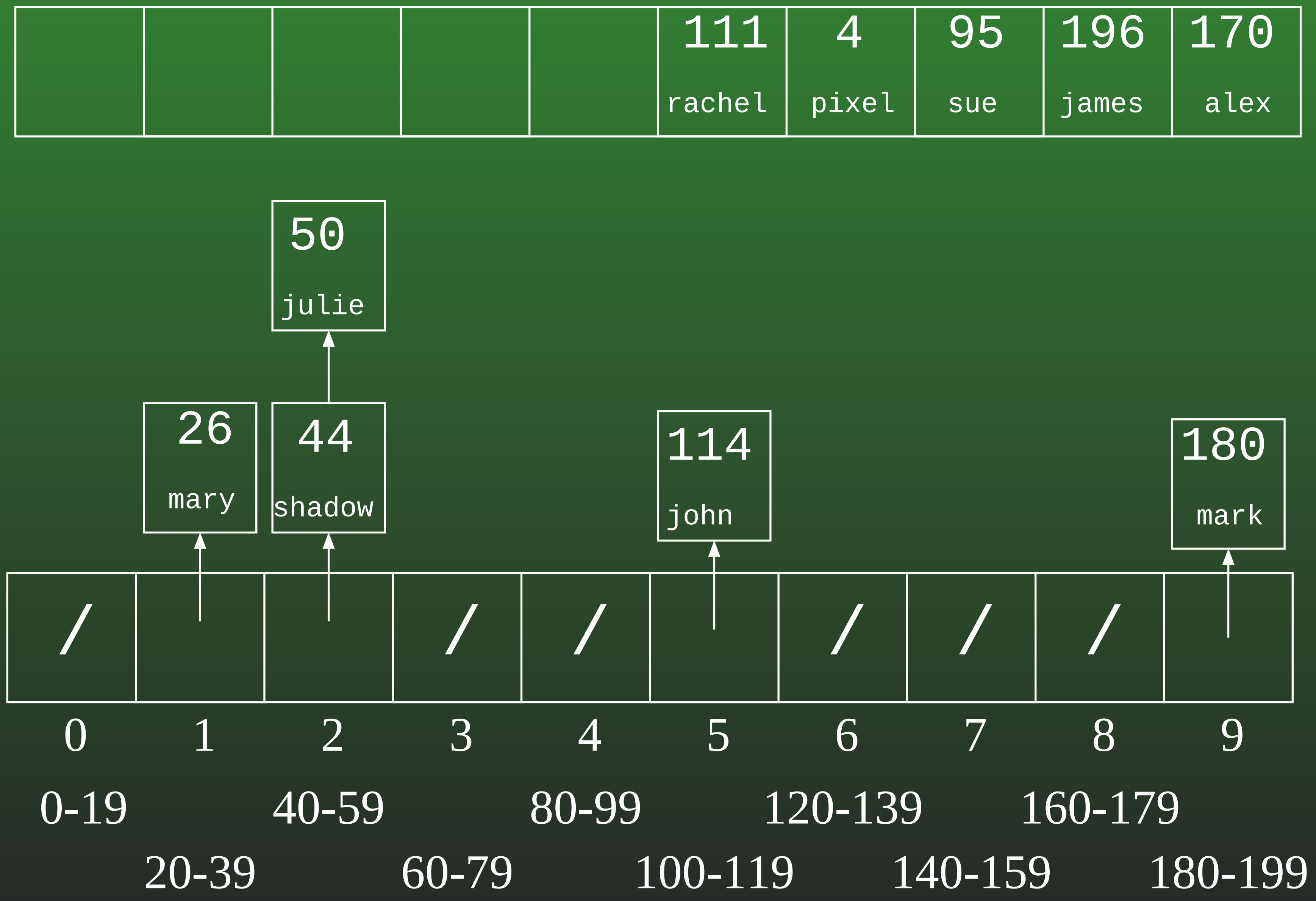


				44	111	4	95	196	170	key
				shadow	rachel	pixel	sue	james	alex	data

26 mary	50 julie	114 john	180 mark

/			/	/		/	/	/	
0	1	2	3	4	5	6	7	8	9
0-19	20-39	40-59	60-79	80-99	100-119	120-139	140-159	160-179	180-199

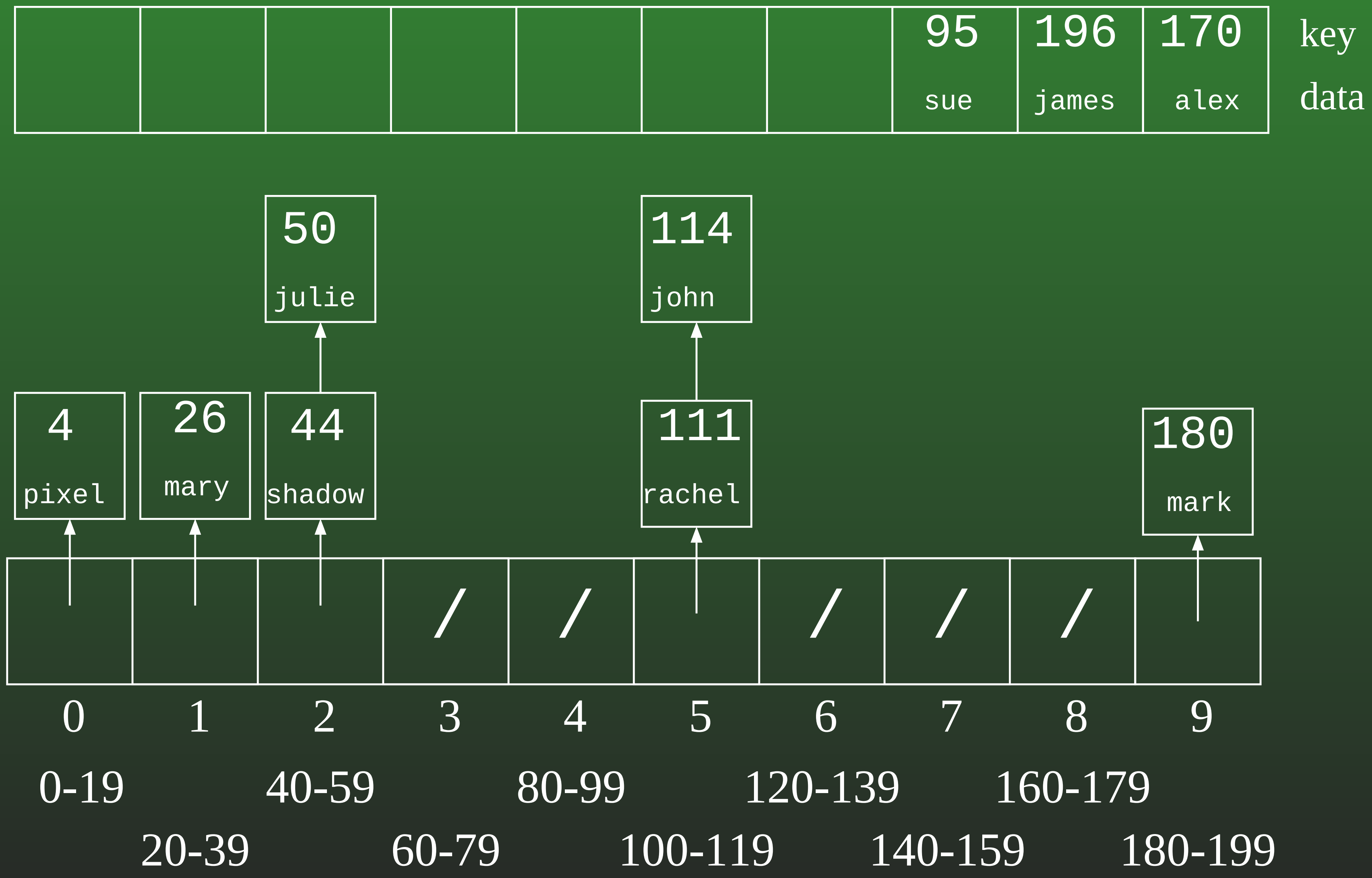
Bucket Sort Example



					111	4	95	196	170	key
					rachel	pixel	sue	james	alex	data

Bucket Sort Example

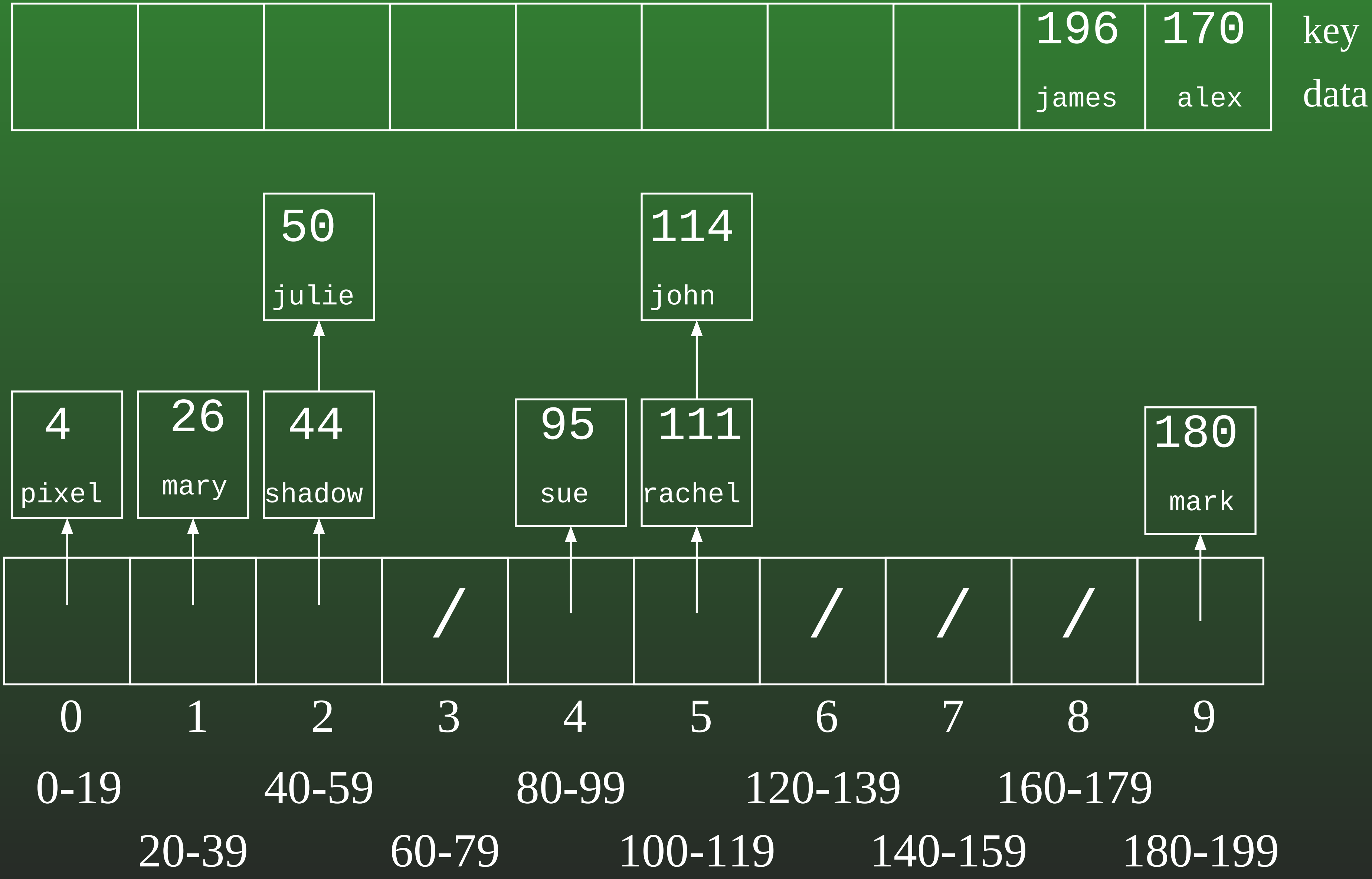
Bucket Sort Example



							95	196	170	key
							sue	james	alex	data

		50 julie			114 john				
4 pixel	26 mary	44 shadow			111 rachel				180 mark
			/	/		/	/	/	
0	1	2	3	4	5	6	7	8	9
0-19	20-39	40-59	60-79	80-99	100-119	120-139	140-159	160-179	180-199

12-41: Bucket Sort Example



196 james
170 alex
key
data

50 julie
114 john

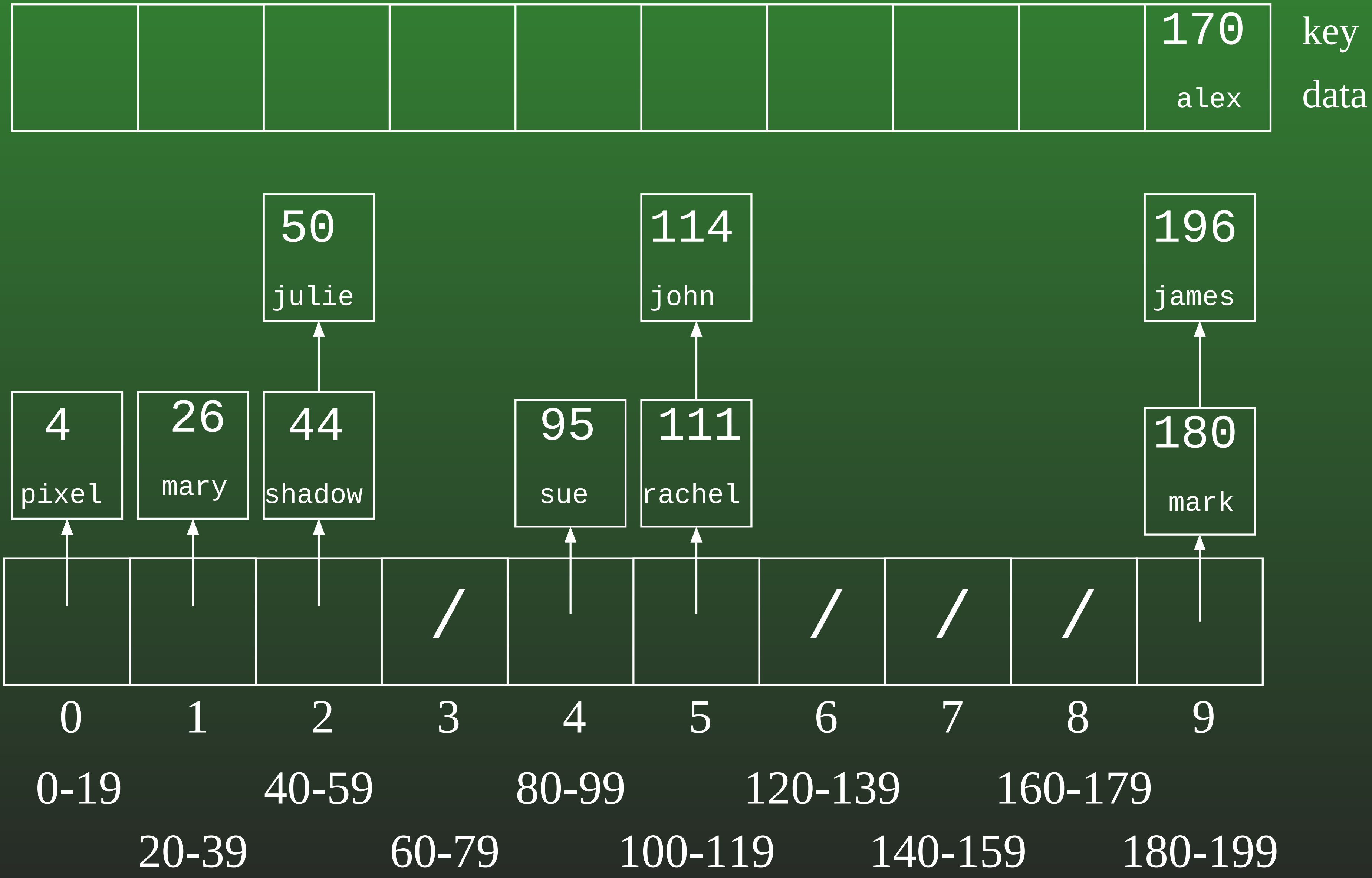
4 pixel
26 mary
44 shadow
95 sue
111 rachel
180 mark

/ / / /

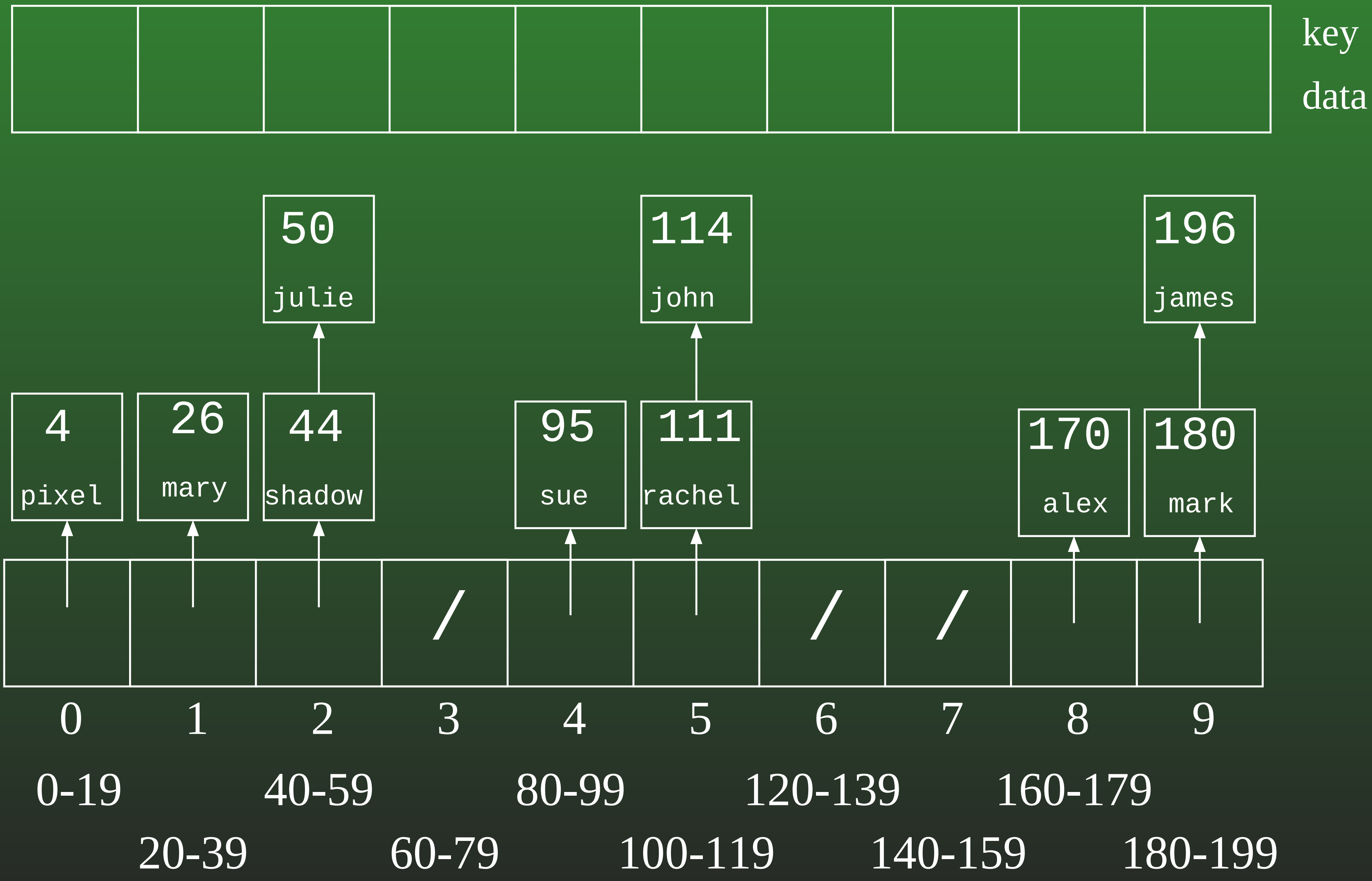
0 1 2 3 4 5 6 7 8 9

0-19
20-39
40-59
60-79
80-99
100-119
120-139
140-159
160-179
180-199

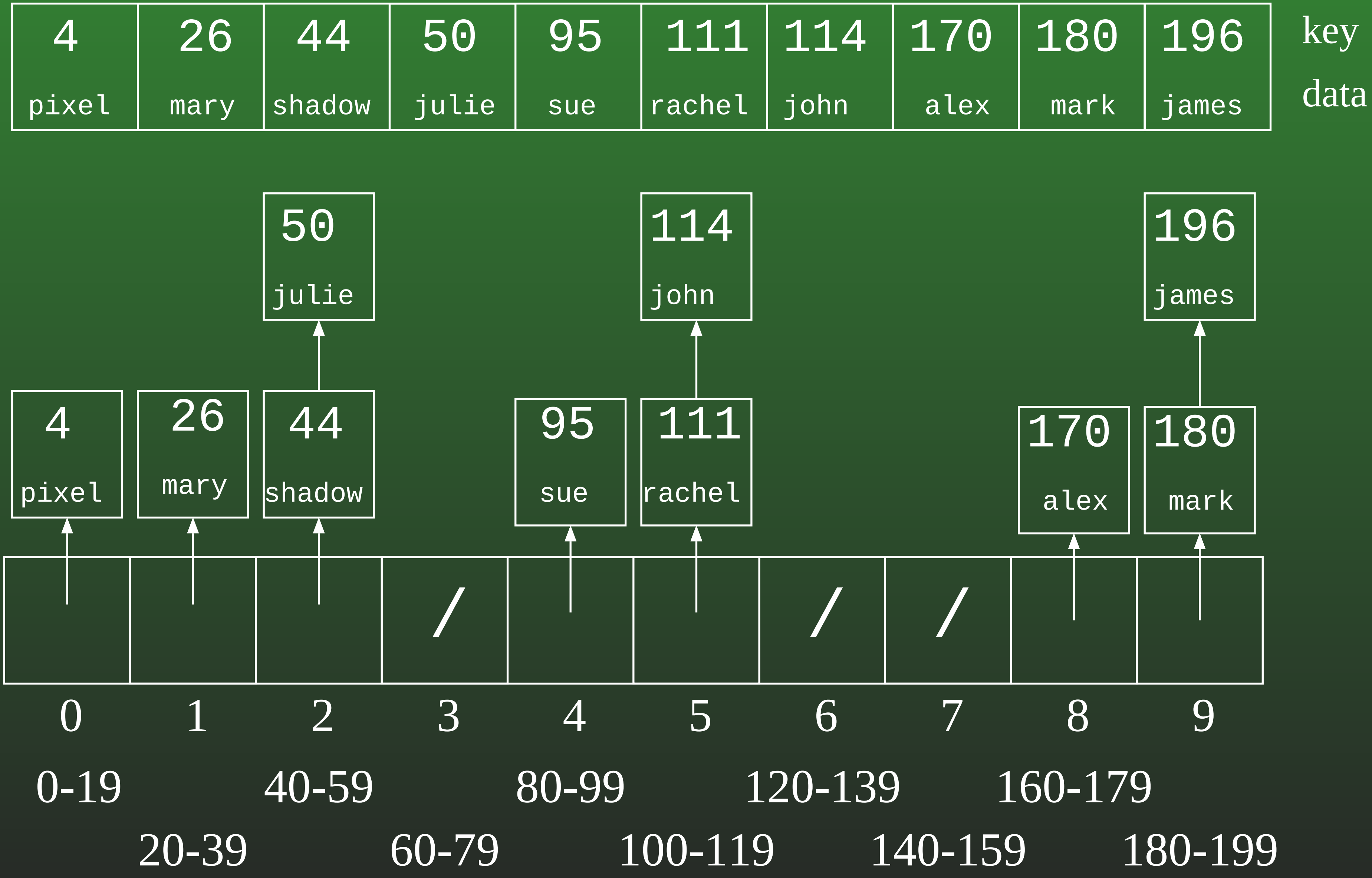
12-42: Bucket Sort Example



Bucket Sort Example



12-44: Bucket Sort Example



Counting Sort Revisited

- We’re going to look at counting sort again
- For the moment, we will assume that our array is indexed from $1 \ldots n$ (where n is the number of elements in the list) instead of being indexed from $0 \ldots n - 1$, to make the algorithm easier to understand
- Later, we will go back and change the algorithm to allow for an index between $0 \ldots n - 1$

Counting Sort Revisited

- Create the array $C[]$, such that $C[i] =$ # of times key i appears in the array.
- Modify $C[]$ such that $C[i] =$ the *index* of key i in the sorted array. (assume no duplicate keys, for now)
- If $x \not\in A$, we don't care about $C[x]$

Counting Sort Revisited

- Create the array $C[]$, such that $C[i] =$ # of times key i appears in the array.
- Modify $C[]$ such that $C[i] =$ the *index* of key i in the sorted array. (assume no duplicate keys, for now)
- If $x \notin A$, we don't care about $C[x]$

```
for(i=1; i<C.length; i++)
   C[i] = C[i] + C[i-1];
```

- Example: 3 1 2 4 9 8 7

Counting Sort Revisited

- Once we have a modified C, such that $C[i] =$ index of key i in the array, how can we use C to sort the array?

Counting Sort Revisited

- Once we have a modified C, such that $C[i] =$ index of key i in the array, how can we use C to sort the array?

```
for (i=1; i <= n; i++)
   B[C[A[i].key()]] = A[i];
for (i=1; i <= n; i++)
   A[i] = B[i];
```

- Example: 3 1 2 4 9 8 7

12-50: Counting Sort & Duplicates

- If a list has duplicate elements, and we create C as before:

```
for(i=1; i <= n; i++)
   C[A[i].key()]++;
for(i=1; i < C.length; i++)
  C[i] = C[i] + C[i-1];
```

What will the value of $C[i]$ represent?

Counting Sort & Duplicates

- If a list has duplicate elements, and we create C as before:

```
for(i=1; i <= n; i++)
   C[A[i].key()]++;
for(i=1; i < C.length; i++)
  C[i] = C[i] + C[i-1];
```

  What will the value of $C[i]$ represent?

  - The *last* index in A where element i could appear.

(Almost) Final Counting Sort

```
for(i=1; i <= n; i++)
   C[A[i].key()]++;
for(i=1; i < C.length; i++)
  C[i] = C[i] + C[i-1];

for (i=1; i <= n; i++) {
   B[C[A[i].key()]] = A[i];
   C[A[i].key()]--;
}
for (i=1; i <= n; i++)
   A[i] = B[i];
```

- Example: 3 1 2 4 2 2 9 1 6

(Almost) Final Counting Sort

```
for(i=1; i <= n; i++)
   C[A[i].key()]++;
for(i=1; i<C.length; i++)
  C[i] = C[i] + C[i-1];

for (i=1; i <= n; i++) {
   B[C[A[i].key()]] = A[i];
   C[A[i].key()]--;
}
for (i=1; i <= n; i++)
   A[i] = B[i];
```

- Example: 3 1 2 4 2 2 9 1 6
- Is this a Stable sorting algorithm?

(Almost) Final Counting Sort

```
for(i=1; i <= n; i++)
   C[A[i].key()]++;
for(i=1; i < C.length; i++)
  C[i] = C[i] + C[i-1];

for (i = n; i>=1; i++) {
   B[C[A[i].key()]] = A[i];
   C[A[i].key()]--;
}

for (i=1; i < n; i++)
   A[i] = B[i];
```

- How would we change this algorithm if our arrays were indexed from $0 \ldots n-1$ instead of $1 \ldots n$?

Final (!) Counting Sort

```
for(i=0; i < A.length; i++)
  C[A[i].key()]++;
for(i=1; i < C.length; i++)
  C[i] = C[i] + C[i-1];

for (i=A.length - 1; i>=0; i++) {
   C[A[i].key()]--;
   B[C[A[i].key()]] = A[i];
}

for (i=0; i < A.length; i++)
   A[i] = B[i];
```

Radix Sort

- Sort a list of numbers one digit at a time
    - Sort by 1st digit, then 2nd digit, etc
- Each sort can be done in linear time, using counting sort
- First Try: Sort by most significant digit, then the next most significant digit, and so on
    - Need to keep track of a lot of sublists

Radix Sort

Second Try:

- Sort by *least significant* digit first
- Then sort by next-least significant digit, using a Stable sort

  . . .
- Sort by most significant digit, using a Stable sort

At the end, the list will be completely sorted. Why?

Radix Sort

- If (most significant digit of x) < (most significant digit of y),

  then x will appear in A before y.

Radix Sort

- If (most significant digit of x) < (most significant digit of y),

  then x will appear in A before y.
  - Last sort was by the most significant digit

Radix Sort

- If (most significant digit of x) < (most significant digit of y),

  then x will appear in A before y.
  - Last sort was by the most significant digit
- If (most significant digit of x) = (most significant digit of y) and

  (second most significant digit of x) < (second most significant digit of y),

  then x will appear in A before y.

Radix Sort

- If (most significant digit of x) <
  (most significant digit of y),

  then x will appear in A before y.
  - Last sort was by the most significant digit
- If (most significant digit of x) =
  (most significant digit of y) and

  (second most significant digit of x) <
  (second most significant digit of y),

  then x will appear in A before y.
  - After next-to-last sort, x is before y. Last sort does not change relative order of x and y

Radix Sort

Original List

982	414	357	495	500	904	645	777	716	637	149	913	817	493	730	331	201

Sorted by Least Significant Digit

$50\underline{0}$	$73\underline{0}$	$33\underline{1}$	$20\underline{1}$	$98\underline{2}$	$49\underline{3}$	$91\underline{3}$	$41\underline{4}$	$90\underline{4}$	$64\underline{5}$	$49\underline{5}$	$71\underline{6}$	$35\underline{7}$	$77\underline{7}$	$63\underline{7}$	$81\underline{7}$	$14\underline{9}$

Sorted by Second Least Significant Digit

$5\underline{0}0$	$2\underline{0}1$	$9\underline{0}4$	$9\underline{1}3$	$4\underline{1}4$	$7\underline{1}6$	$8\underline{1}7$	$7\underline{3}0$	$3\underline{3}1$	$6\underline{3}7$	$6\underline{4}5$	$1\underline{4}9$	$3\underline{5}7$	$7\underline{7}7$	$9\underline{8}2$	$4\underline{9}3$	$4\underline{9}5$

Sorted by Most Significant Digit

$\underline{1}49$	$\underline{2}01$	$\underline{3}31$	$\underline{3}57$	$\underline{4}14$	$\underline{4}93$	$\underline{4}95$	$\underline{5}00$	$\underline{6}37$	$\underline{6}45$	$\underline{7}16$	$\underline{7}30$	$\underline{7}77$	$\underline{8}17$	$\underline{9}04$	$\underline{9}13$	$\underline{9}82$

Radix Sort

- We do not need to use a single digit of the key for each of our counting sorts
    - We could use 2-digit chunks of the key instead
    - Our C array for each counting sort would have 100 elements instead of 10

12-64: Radix Sort

Original List

9823	4376	2493	1055	8502	4333	1673	8442	8035	6061	7004	3312	4409	2338

Sorted by Least Significant Base-100 Digit (last 2 base-10 digits)

8502	7004	4409	3312	9823	4333	8035	2338	8442	1055	6061	1673	4376	2493

Sorted by Most Significant Base-100 Digit (first 2 base-10 digits)

1055	1673	2338	2493	3312	4333	4376	4409	6061	7004	8035	8442	8502	9823

Radix Sort

- “Digit” does not need to be base ten
- For any value r:
    - Sort the list based on (key % r)
    - Sort the list based on ((key / r) % r))
    - Sort the list based on ((key / r^2) % r))
    - Sort the list based on ((key / r^3) % r))

    $\ldots$
    - Sort the list based on ((key / $r^{\log_k(\text{largest value in array})}$) % r))

- Code on other screen