# Week 3
# Advanced Sorting Algorithms
# Linear Data Structures
# Amortization

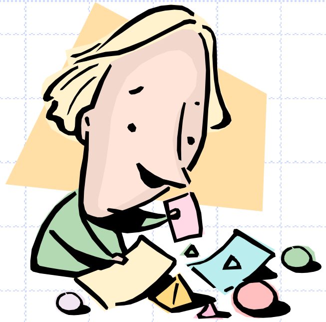
Algorithms and Data Structures

COMP3506/7505

# Week 3 – Sorting & Linear DS

1. Bucket-sort and radix-sort
2. Arrays
3. Linked Lists
4. Extensible Lists and Amortization

# Comparison Sorts



- Sorted order determined by comparing keys

```
if A[k] < A[k+1]:
  # Do some swapping
```

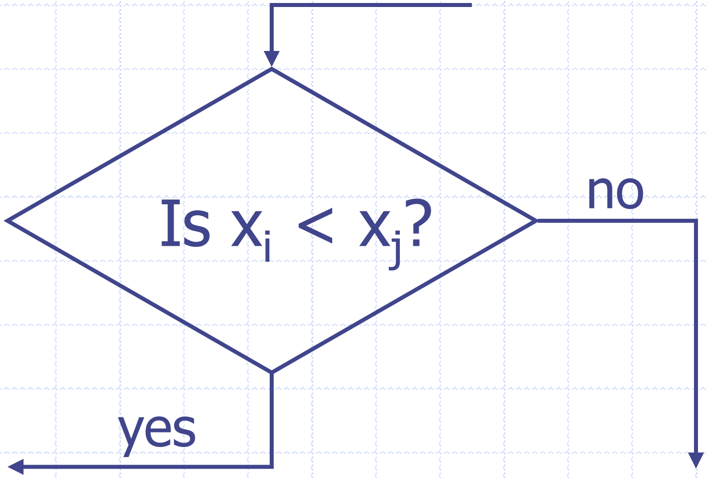
- Examples
  - Bubble, insertion, selection, merge and quick sorts
    - Everything we’ve seen so far!

# “Hang on, last week you said”

- I told you that it doesn’t get any better than *mergesort*
  - At least, in terms of the *number of comparisons* that must be done to sort the input
  - That is, any comparison-based sorting algorithm must have $\Omega(n \log_2 n)$ comparisons in the worst case

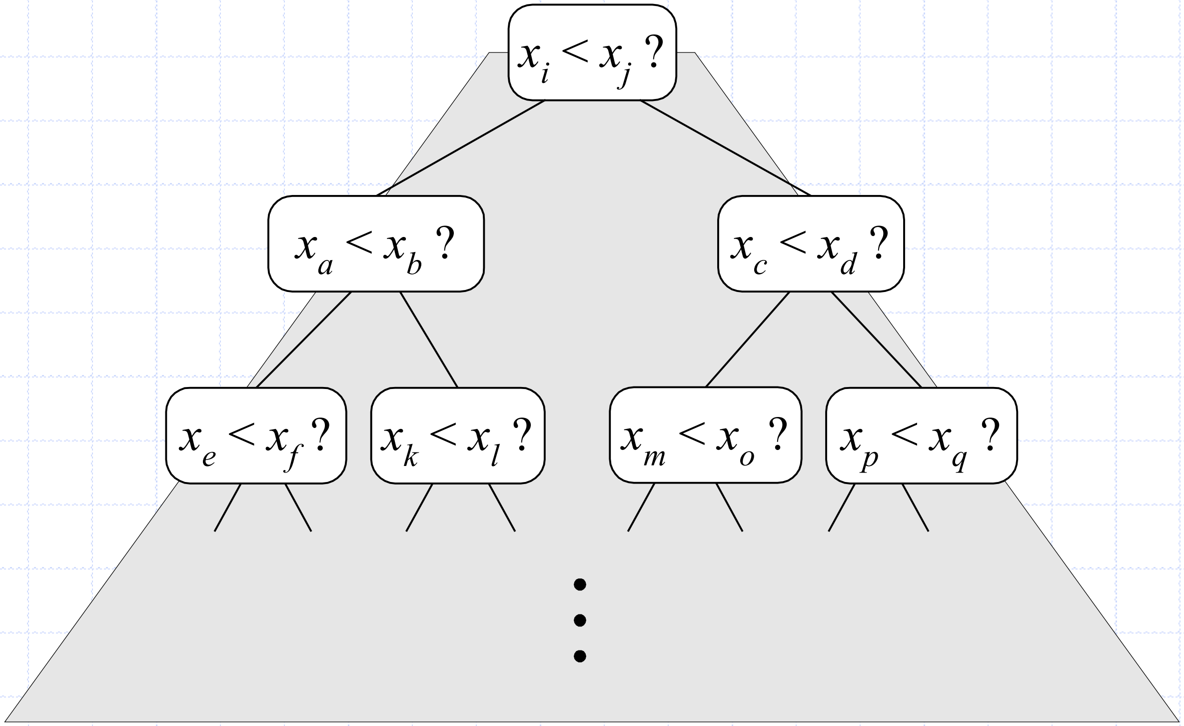
# Determining Performance

- Derive a worst-case lower bound on running time of any algorithm that uses **comparisons** to sort $n$ elements, $x_1$, $x_2$, …, $x_n$



# Counting Comparisons

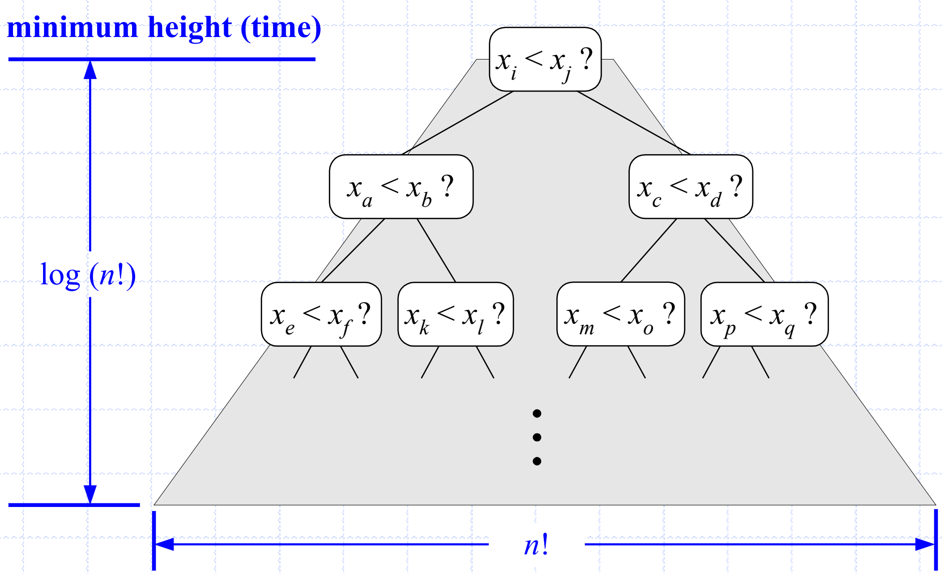
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree



# Think Permutations…

# Decision Tree Height

- Height of decision tree is a lower bound on running time
- Every unique input permutation must lead to a separate leaf output
  - if not, some input …4…5… would have same output ordering as …5…4…, which would be wrong
- There are $n! = 1 \cdot 2 \cdot \ldots \cdot n$ leaves – height is at least $\log(n!)$



# Proof.

$$
\begin{aligned}
2^h \geq n! \Rightarrow h &\geq \log(n!) \\
&= \log(n(n-1)(n-2)\cdots(2)) \\
&= \log n + \log(n-1) + \log(n-2) + \cdots + \log 2 \\
&= \sum_{i=2}^{n} \log i \\
&= \sum_{i=2}^{n/2-1} \log i + \sum_{i=n/2}^{n} \log i \\
&\geq 0 + \sum_{i=n/2}^{n} \log \frac{n}{2} \\
&= \frac{n}{2} \cdot \log \frac{n}{2} \\
&= \Omega(n \log n)
\end{aligned}
$$

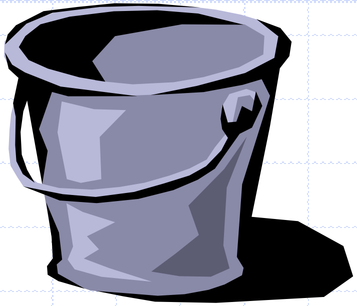
# Lower Bound

- Comparison-based sorting algorithms takes at least $\log(\boldsymbol{n}!)$ time
- ∴ any such algorithm takes at least
  - $\log(n!) \geq \log\left(\frac{n}{2}\right)^{n/2} = (n/2)\log(n/2)$
- Any comparison-based sorting algorithm must run in $\Omega(\boldsymbol{n} \log \boldsymbol{n})$ time
  - merge and heap sorts are asymptotically optimal
    - no other comparison sorts are faster by more than a constant factor

# Can We Do Better?

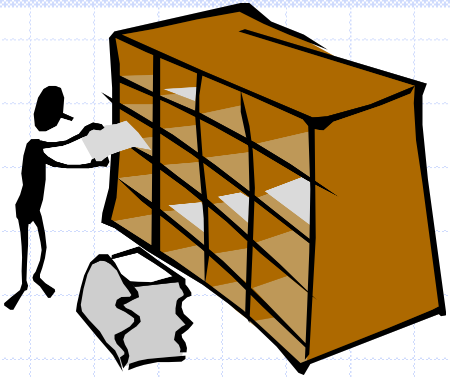
- Do we have to compare every key to sort data?
- Do we have to compare anything???
- Idea: Use a data structure to help us…

# Bucket-Sort



- Let $S$ be a list of $n$ (key, element) items with keys in the range $[0, N-1]$
- Bucket-sort uses the keys as indices into an auxiliary array $B$ of lists (buckets)
  - Phase 1: Empty list $S$ by moving each entry $(k, o)$ into its bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ to the end of list $S$

# Example

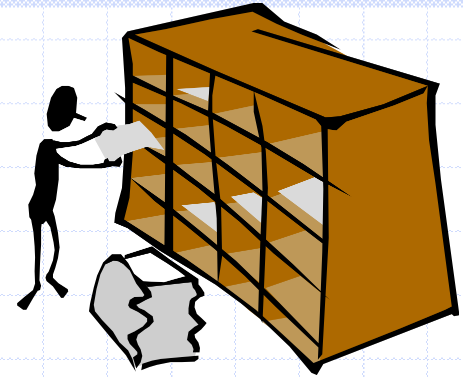


- Key range [0, 9]

$(7, d)$ — $(1, c)$ — $(3, a)$ — $(7, g)$ — $(3, b)$ — $(7, e)$

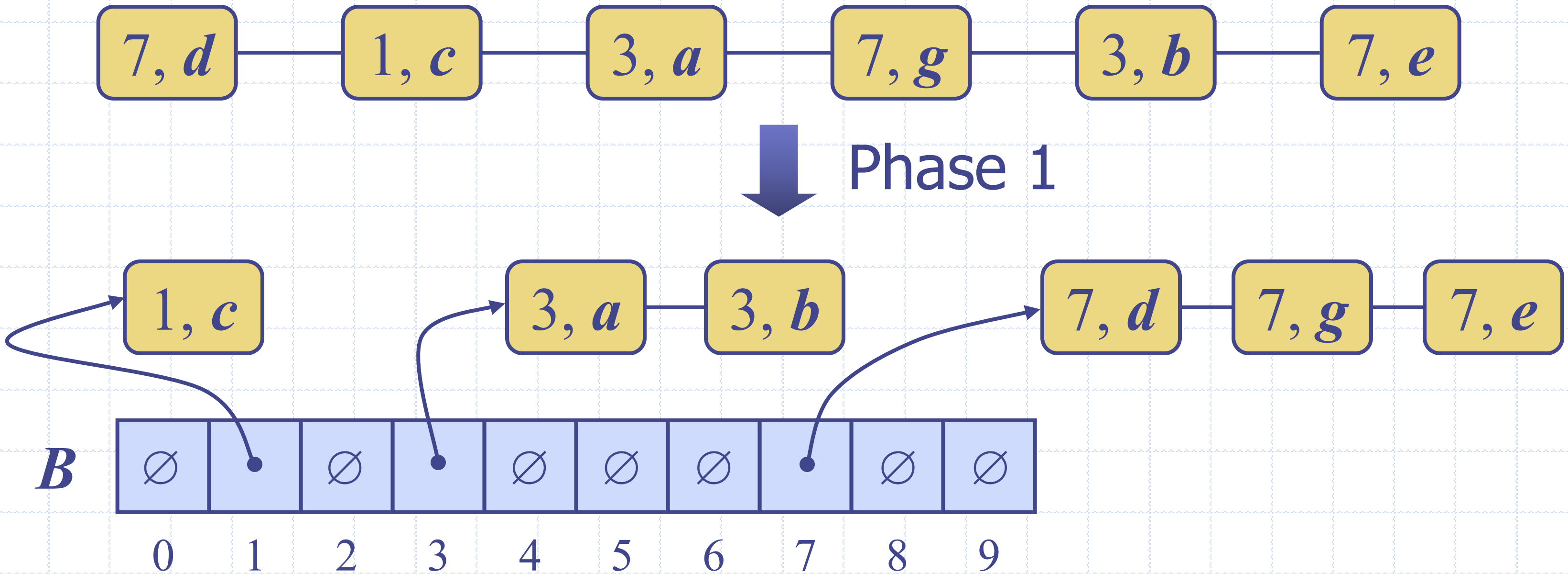
***B***

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

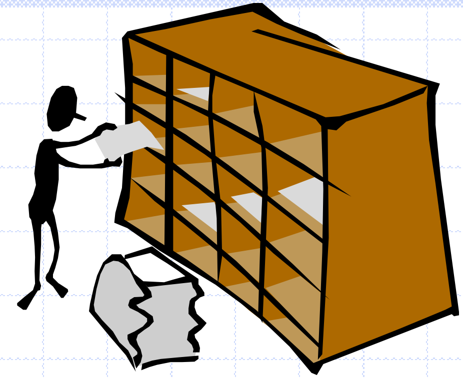


# Example

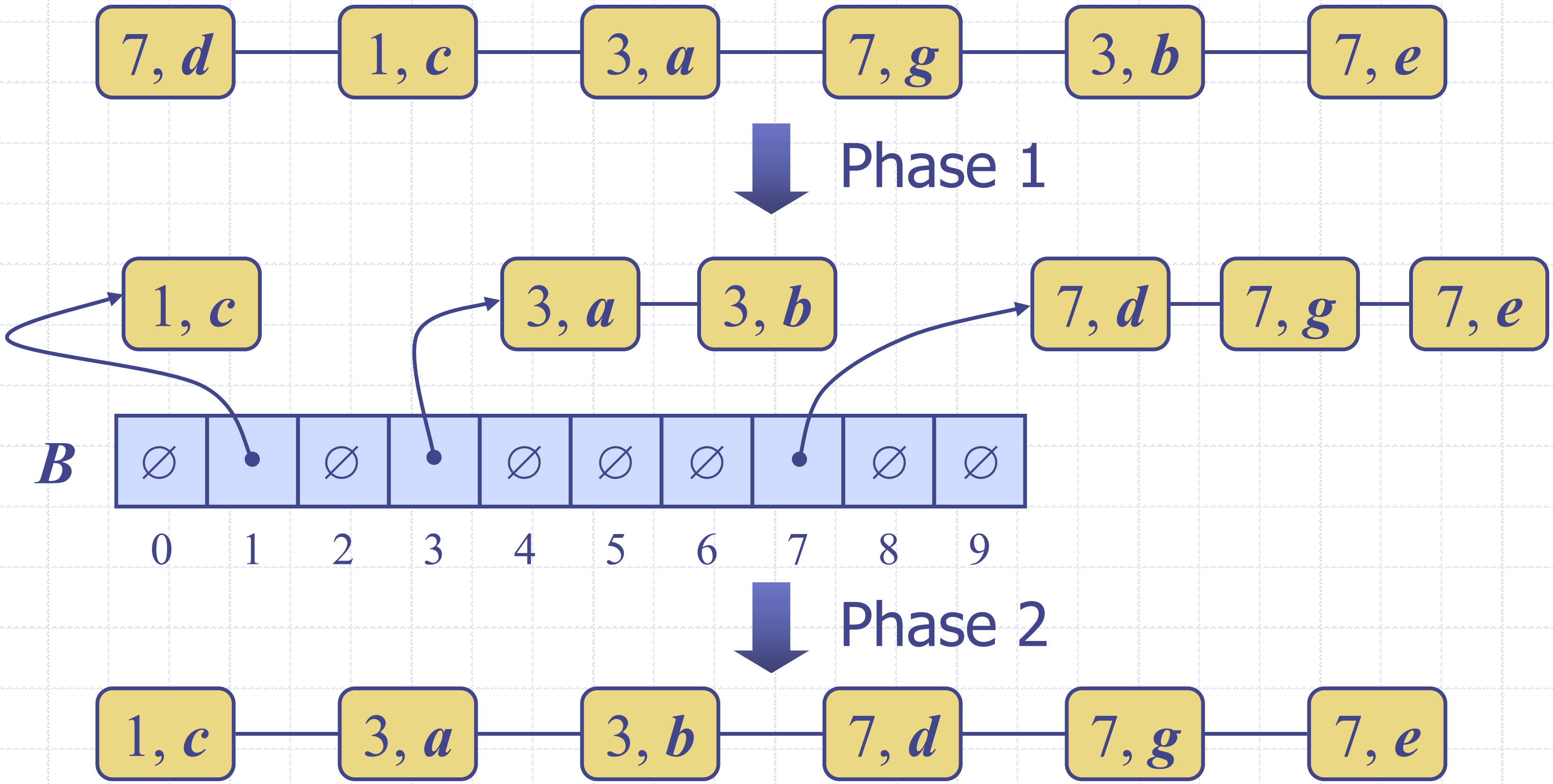
- Key range $[0, 9]$



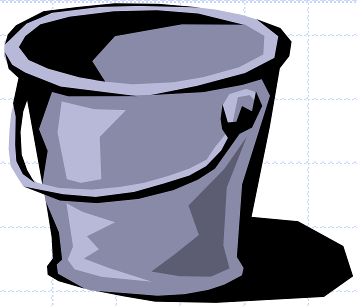
# Example



- Key range [0, 9]



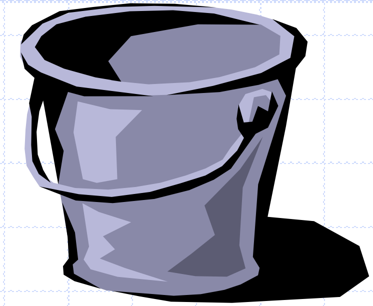
# Bucket-Sort



**Algorithm** ***bucketSort***($S$):
  **Input:** sequence $S$ of $n$ entries with integer keys in the range $[0, N-1]$
  **Output:** sequence $S$ sorted in nondecreasing order of the keys

$B \leftarrow$ array of $N$ empty sequences
**for each** entry $e$ **in** $S$ **do**
  $k \leftarrow$ key of $e$
  remove $e$ from $S$
  insert $e$ at the end of bucket $B[k]$
**for** $i \leftarrow 0$ **to** $N-1$ **do**
  **for each** entry $e$ **in** $B[i]$ **do**
    remove $e$ from $B[i]$
    insert $e$ at the end of $S$

# Bucket-Sort



**Algorithm** ***bucketSort***$(S)$:
  **Input:** sequence $S$ of $n$ entries with integer keys in the range $[0, N-1]$
  **Output:** sequence $S$ sorted in nondecreasing order of the keys

```
B ← array of N empty sequences
for each entry e in S do
    k ← key of e
    remove e from S
    insert e at the end of bucket B[k]
for i ← 0 to N−1 do
    for each entry e in B[i] do
        remove e from B[i]
        insert e at the end of S
```

- Analysis
  - Initialising the bucket array takes $O(N)$ time
  - Phase 1 takes $O(n)$ time
  - Phase 2 takes $O(n + N)$ time
- Bucket-Sort takes $O(n + N)$ time

# Properties and Extensions

- Stable Sort Property
  - Relative order of any two items with the same key is preserved after the execution of the algorithm

$(7, d)$ — $(1, c)$ — $(3, a)$ — $(7, g)$ — $(3, b)$ — $(7, e)$

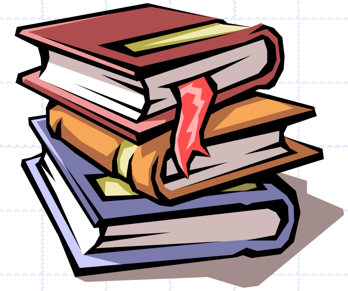
↓ Stable sort

$(1, c)$ — $(3, a)$ — $(3, b)$ — $(7, d)$ — $(7, g)$ — $(7, e)$

# Lexicographic Order

- $d$-tuple is a sequence of $d$ keys $(k_1, k_2, \ldots, k_d)$, where key $k_i$ is said to be the $i$-th dimension of the tuple
  - e.g. Cartesian coordinates of a point in space are a 3-tuple
  - e.g. ant < apple

# Lexicographic Order



- Lexicographic order of two $d$-tuples is recursively defined as follows

$$(x_1, x_2, \ldots, x_d) < (y_1, y_2, \ldots, y_d)$$
$$\Leftrightarrow$$
$$x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \ldots, x_d) < (y_2, \ldots, y_d)$$

- i.e. tuples are compared by the first dimension, then by the second dimension, etc.
  - $(2, 1, 4) < (3, 2, 5)$ since $2 < 3$
  - $(2, 1, 4) < (2, 2, 5)$ since $2 = 2$ and $1 < 2$
  - $(2, 1, 4) < (2, 1, 5)$ since $2 = 2$, $1 = 1$ and $4 < 5$

# Lexicographic-Sort (aka Tuple Sort)

- $C_i$ – comparator that compares two tuples by their $i$-th dimension
- $stableSort(S, C)$ – stable sorting algorithm that uses comparator $C$
- *lexicographicSort* sorts a sequence of $d$-tuples in lexicographic order by executing *stableSort*, $d$ times
  - once per dimension
- Runs in $O(d \cdot T(n))$ time
  - where $T(n)$ is the running time of *stableSort*

**Algorithm** *lexicographicSort*($S$)
  **Input** sequence $S$ of $d$-tuples
  **Output** sequence $S$ sorted in lexicographic order

  **for** $i \leftarrow d$ **downto** 1
    $stableSort(S, C_i)$

Example

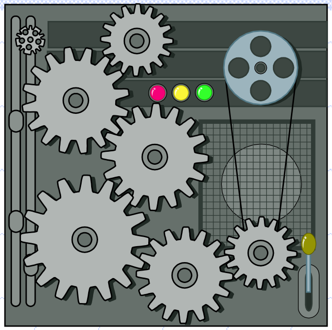
(7,4,6) (5,1,5) (2,4,6) (2,1,4) (3,2,4)

(2,1,4) (3,2,4) (5,1,5) (7,4,6) (2,4,6)

(2,1,4) (5,1,5) (3,2,4) (7,4,6) (2,4,6)

(2,1,4) (2,4,6) (3,2,4) (5,1,5) (7,4,6)

# Radix-Sort



- Specialisation of lexicographic-sort
  - uses bucket-sort as the stable sorting algorithm in each dimension
- Applicable to tuples where the keys in each dimension $i$ are integers in the range $[0, N-1]$
- Runs in time $O(d \cdot (n+N))$

**Algorithm** *radixSort*$(S, N)$
    **Input** sequence $S$ of $d$-tuples such that $(0, \ldots, 0) \le (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \le (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in $S$
    **Output** sequence $S$ sorted in lexicographic order

    **for** $i \leftarrow d$ **downto** 1
        *bucketSort*$(S, N)$

# Radix-Sort for Binary Numbers

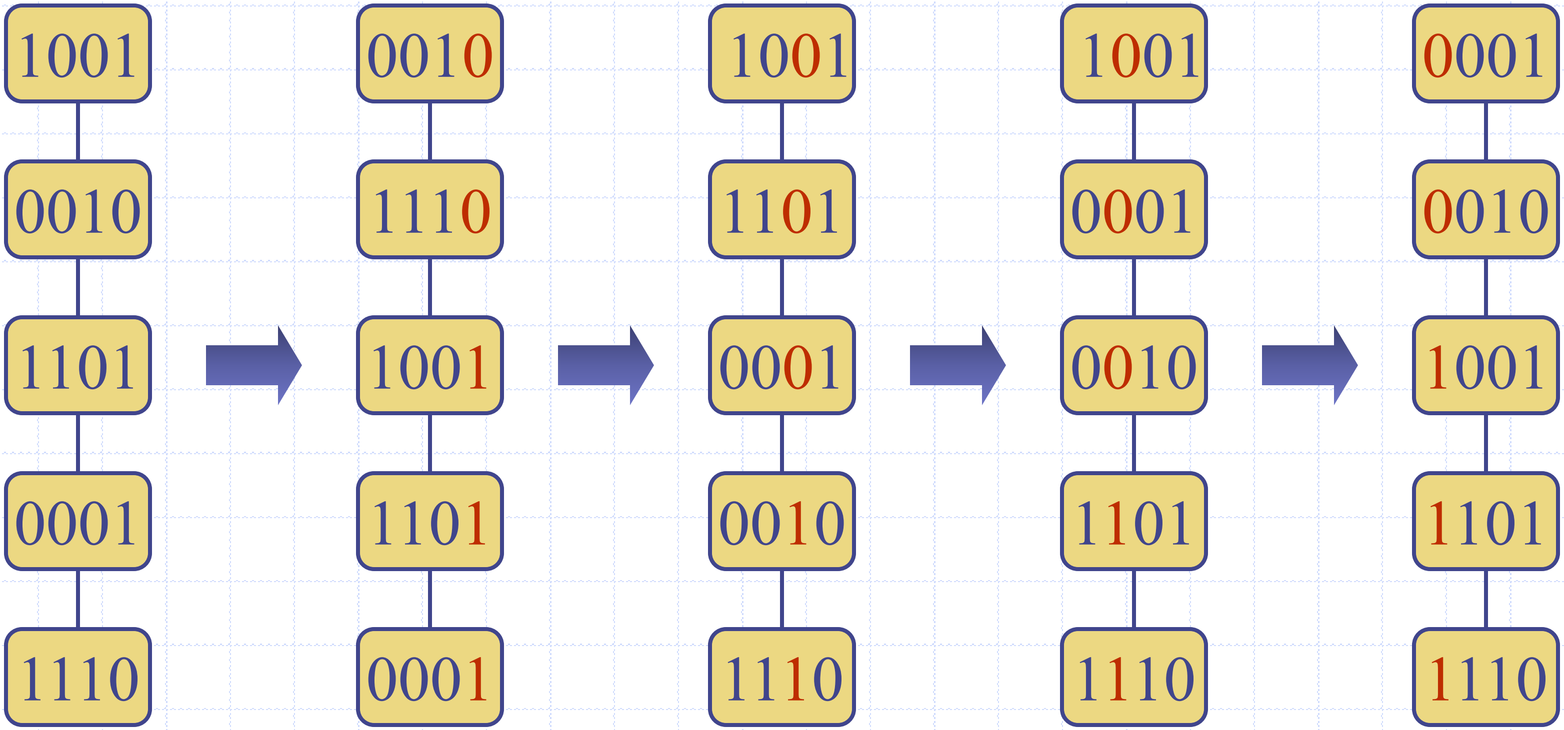
- Consider a sequence of $n$ $b$-bit integers
  - $x = x_{b-1} \dots x_1 x_0$
- Represent each element as a $b$-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$
- Runs in $b\cdot(n+2)$ or $O(b\cdot n)$ time

**Algorithm** *binaryRadixSort*($S$)
  **Input** sequence $S$ of $b$-bit integers
  **Output** sequence $S$ sorted

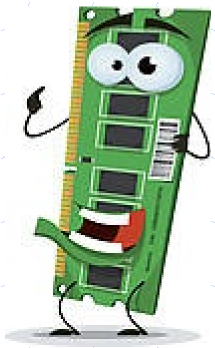
  replace each element $x$ of $S$ with item $(0, x)$
  **for** $i \leftarrow 0$ **to** $b - 1$
    replace key $k$ of item $(k, x)$ with bit $x_i$ of $x$
    *bucketSort*($S$, 2)

# Example

- Sorting a sequence of 4-bit integers



# Memory Usage



- Original sequence and bucket array
  - $O(n + \mathrm{N})$
- Sort: 10, 999, 3, 100 000 000, 20

- Bucket Sort
  - $O(5 + 100\,000\,000)$
- (Binary) Radix Sort
  - $O(5 + 2)$
- (Bytewise) Radix Sort
  - $O(5 + 256)$

# Week 3 – Sorting & Linear DS

1. Bucket-sort and radix-sort
2. Arrays
3. Linked Lists
4. Extensible Lists and Amortization

# General Linear Structures

A linear structure is one whose elements can be seen as being in a sequence. That is, one element follows the next.

- Lists
- Stacks
- Queues
- Vectors

- Recall the static sequence from Lec 1

# Static Sequence ADT

- Given a list of items X in some order: $x_1, x_2, \ldots, x_n$

| | |
|---|---|
| `build(X):` | Make new data structure for items in X |
| `len(X):` | Return *n* |
| `get(i):` | Return the element at position *i* |
| `set(i, x):` | Set $x_i$ to x |

- Note that the way we store the data and compute those functions depend on the *data structure we use*

# Dynamic Sequence ADT

- Given a list of items X in some order: $x_1, x_2, \ldots, x_n$

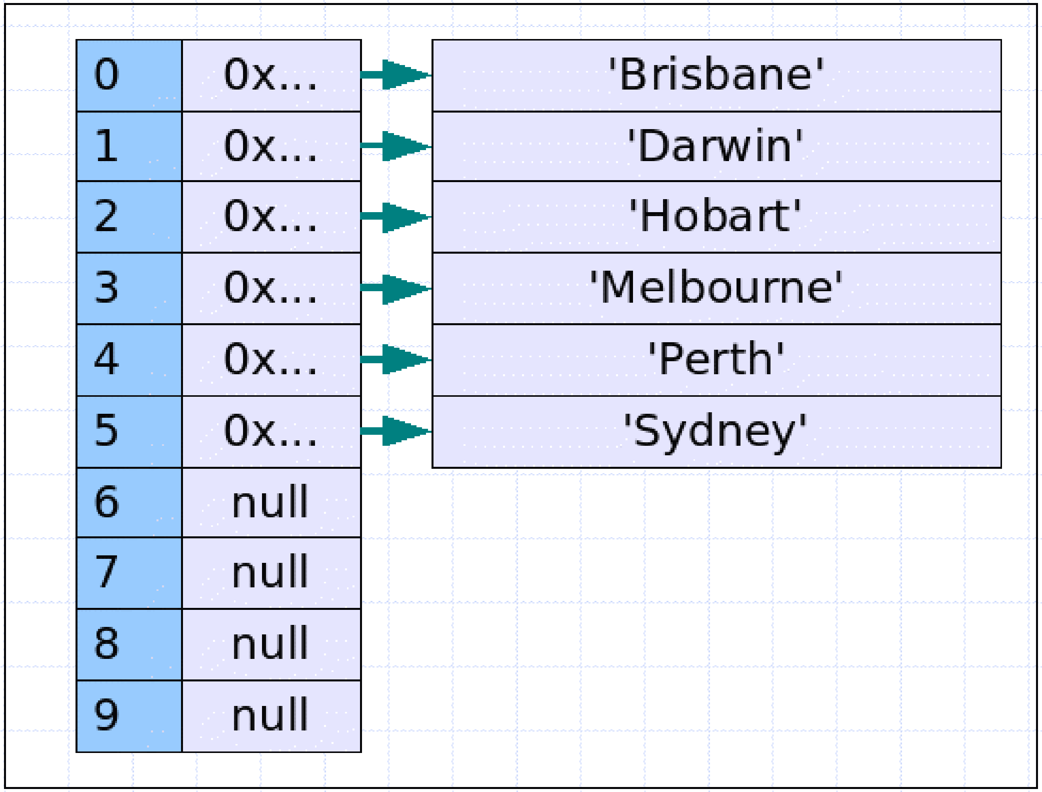
| | |
|---|---|
| `build(X):` | Make new data structure for items in X |
| `len(X):` | Return *n* |
| `get(i):` | Return the element at position *i* |
| `set(i, x):` | Set $x_i$ to x |
| **`add`**`(x):` | Add x to X |

# Implementing Linear Structures

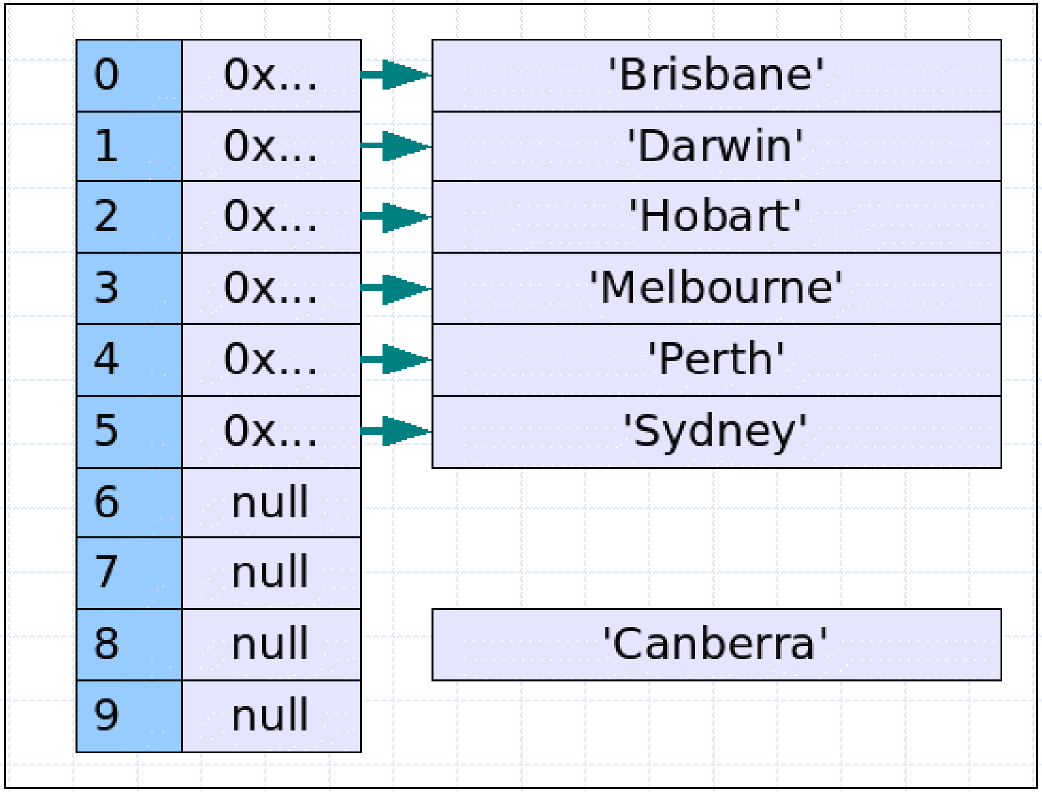
- Arrays: Sequence of consecutive memory cells
- Size must be specified at creation (static!)
- What does it get us?
  - Constant time random access – nice!
  - But what if we want to insert something?
  - What if we need more space?

# Arrays (insert)

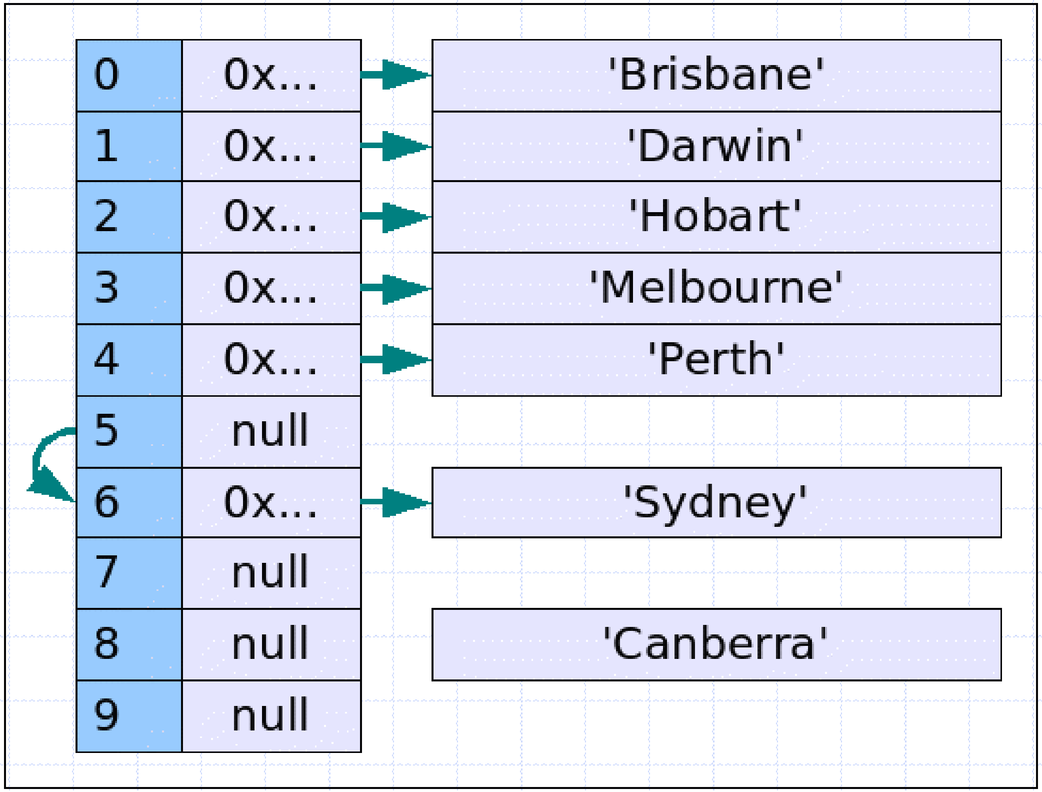
- Insert the value 'Canberra', so that the array maintains sorted order



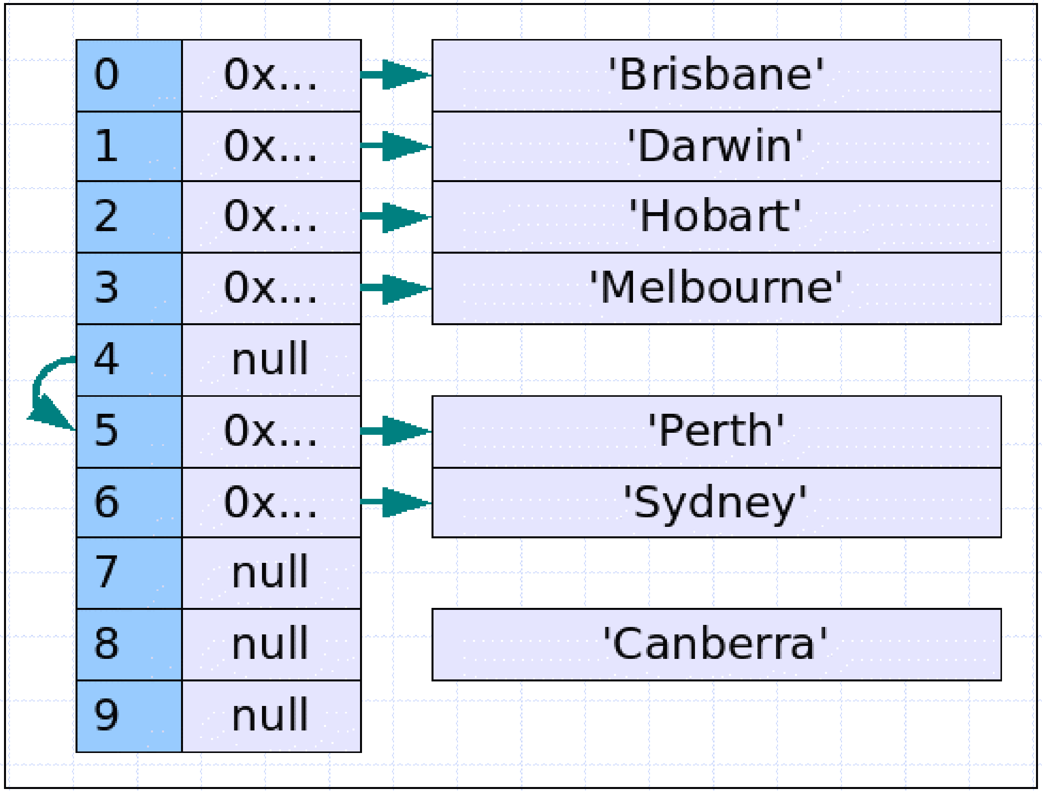
# Arrays (insert)



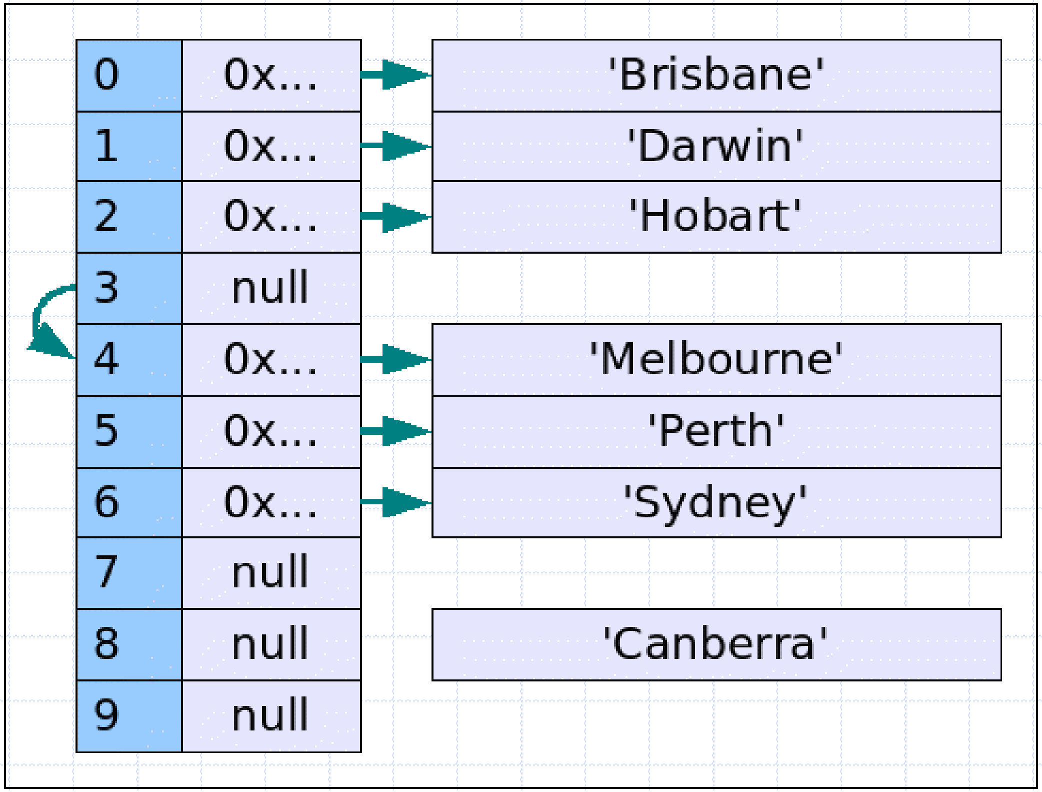
# Arrays (insert)



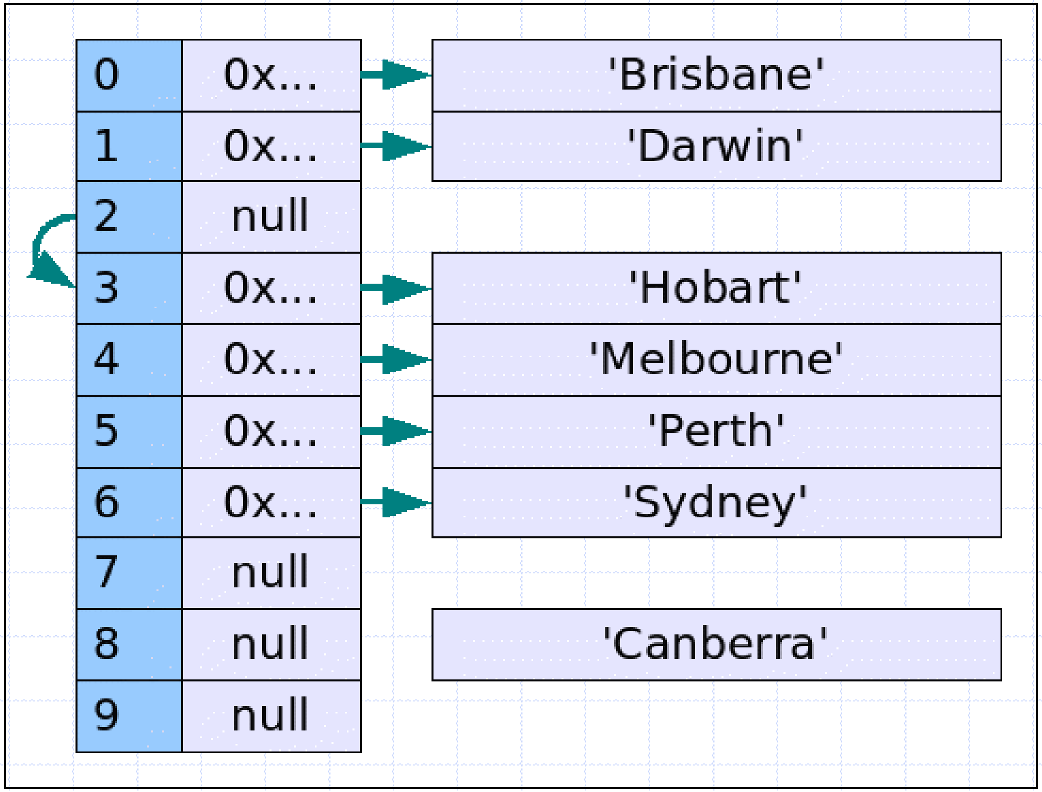
# Arrays (insert)



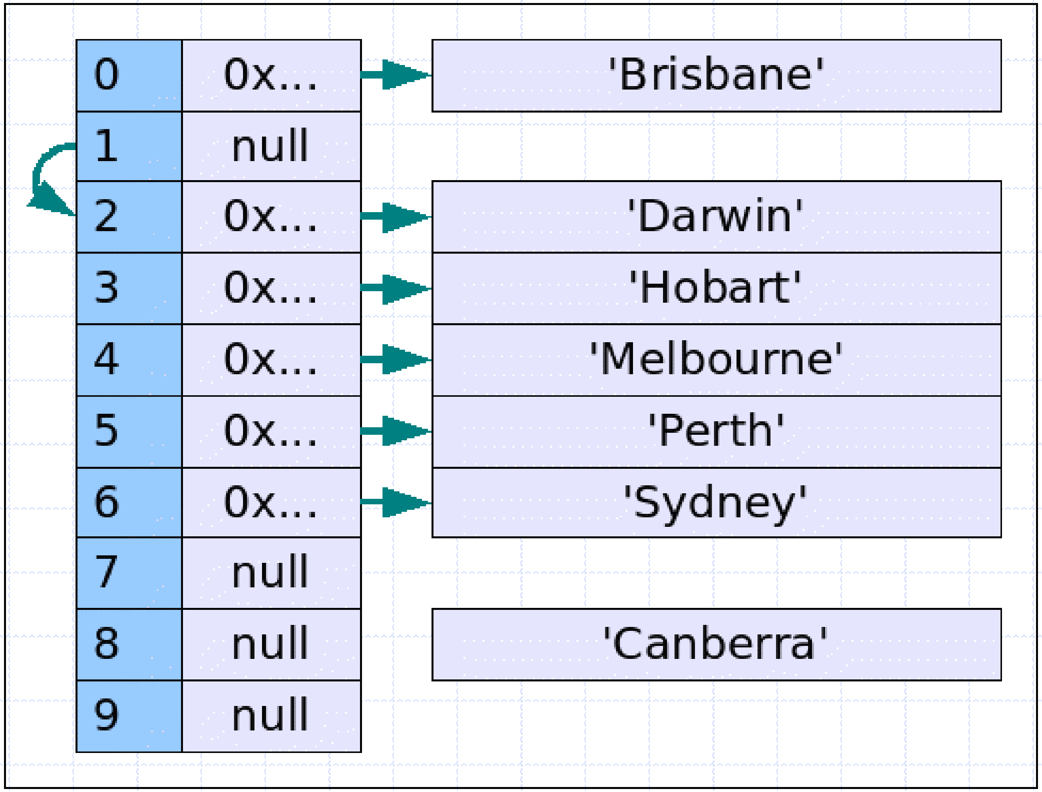
# Arrays (insert)



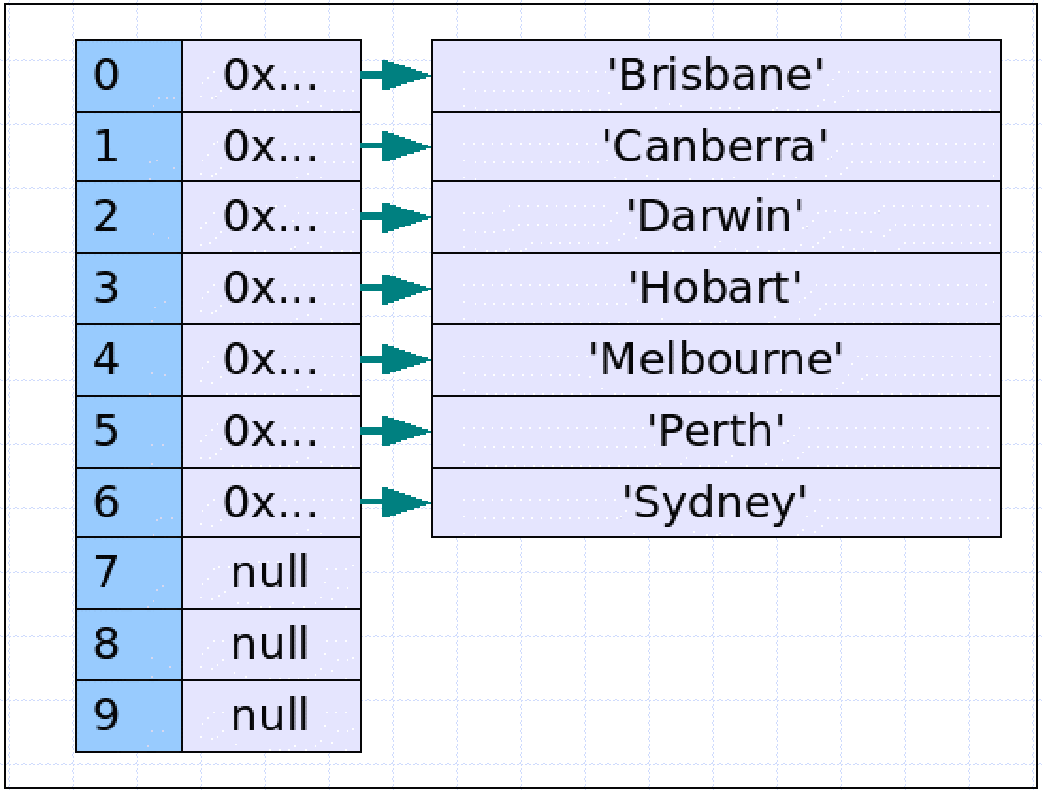
# Arrays (insert)



# Arrays (insert)



# Arrays (insert)



# Array Implementation Efficiency

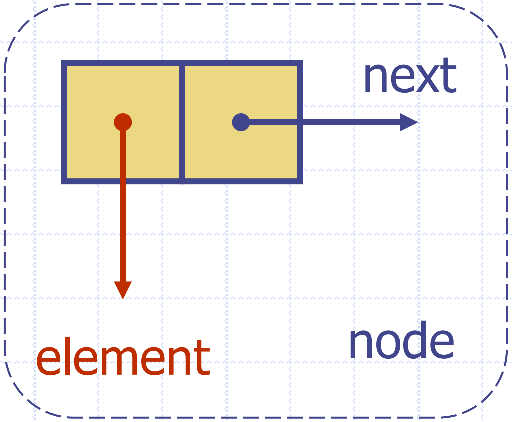
- Accessing an element (by index)
  - O(1)
- Iterating over elements
  - O(n)
- Insert / Delete element
  - O(n)
- Memory usage
  - O(n)

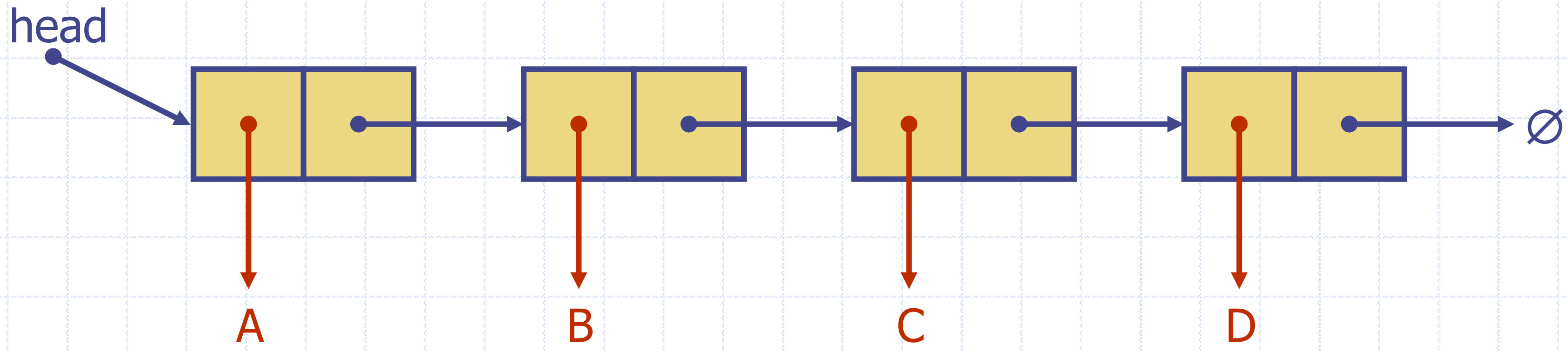
# Week 3 – Sorting & Linear DS

1. Bucket-sort and radix-sort
2. Arrays
3. Linked Lists
4. Extensible Lists and Amortization

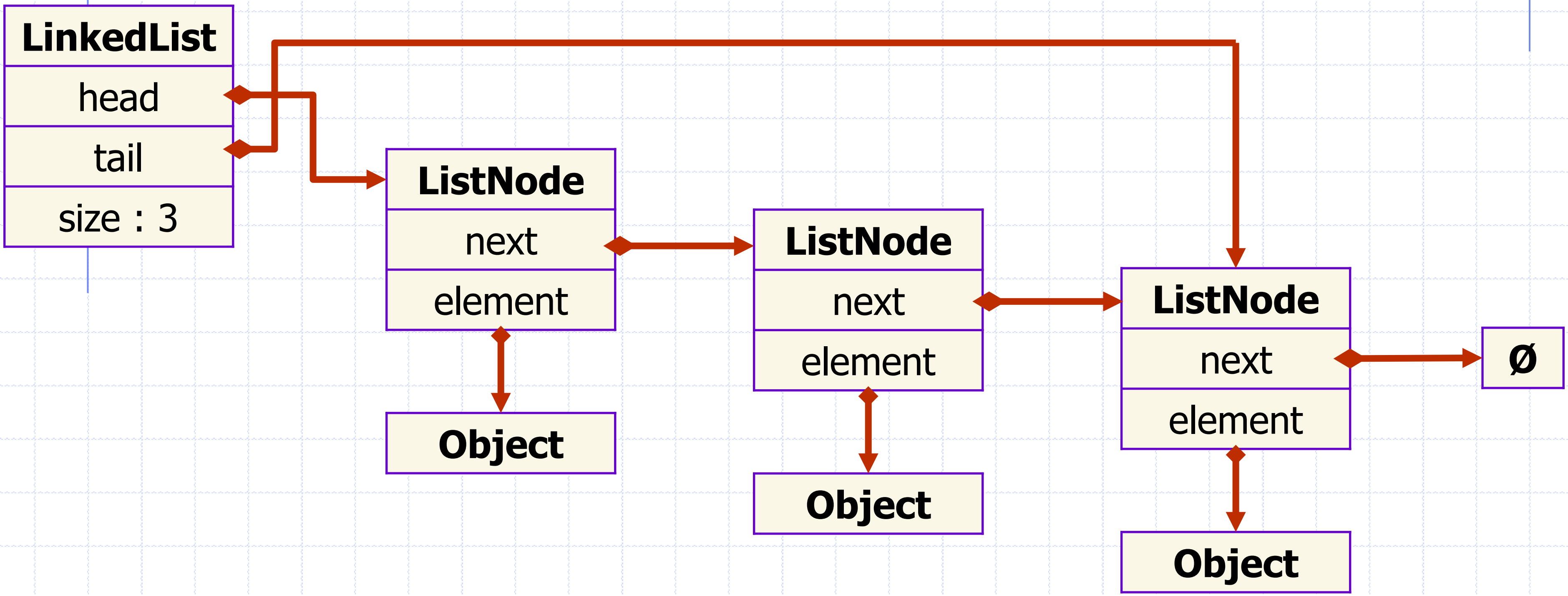
# Singly Linked List

- Concrete data structure
  - sequence of nodes
  - head pointer
- Nodes store
  - element
  - link to next node

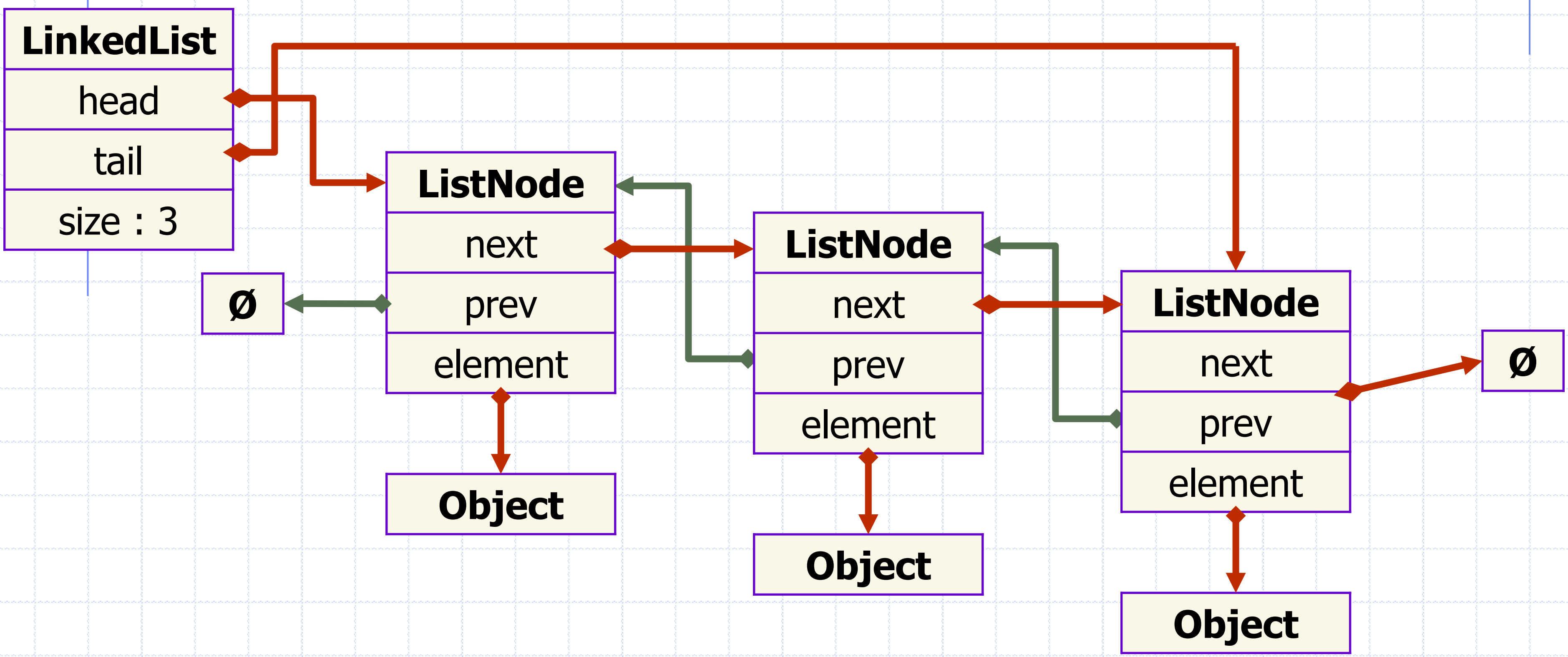


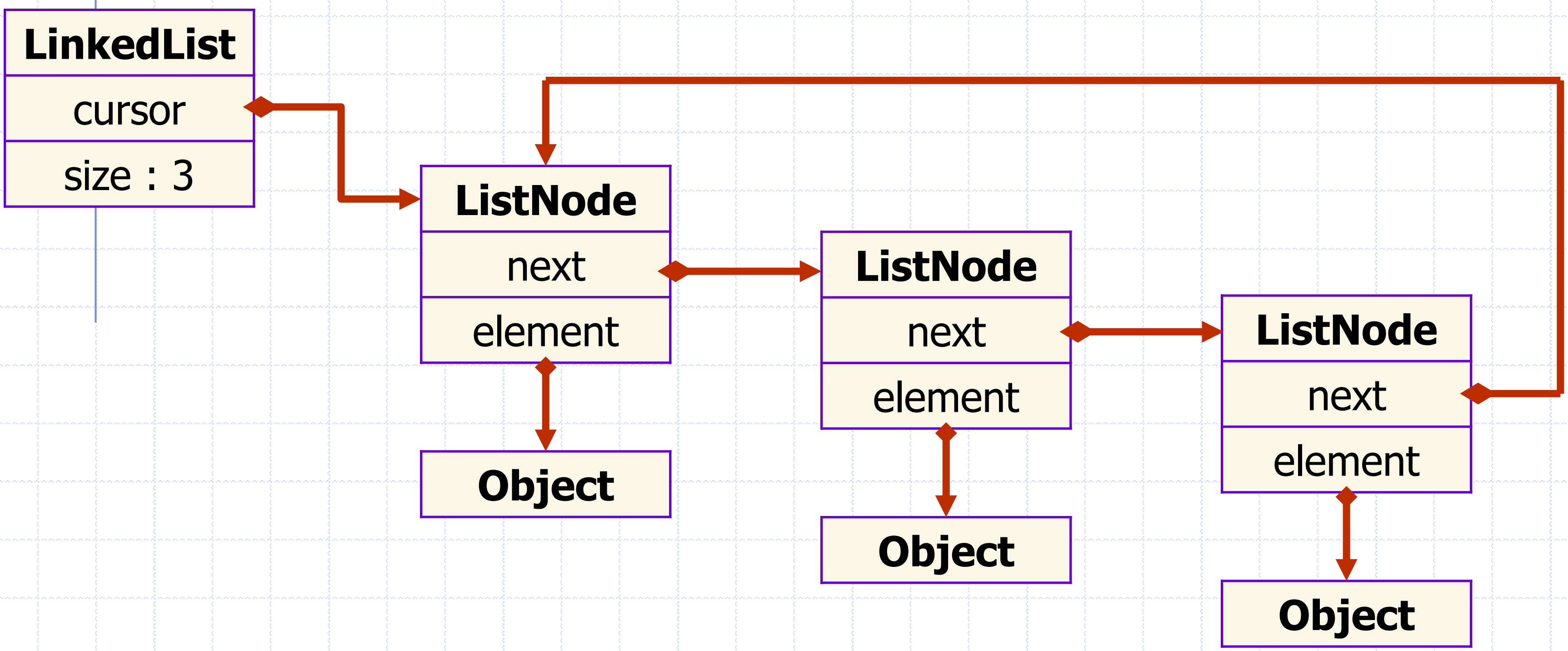
# Singly Linked List



# Doubly Linked List



# Circularly Linked List



# Linked List Sketch

Joel’s note to self: We will draw the in-memory layout of a linked list (word RAM)

# A python linked list

```
class Node:
  def __init__(self, data):
    self.__data = data
    self.__next = None

  def set_data(self, data):
    self.__data = data

  ...

   def get_next(self):
     return self.__next
```

```
class LinkedList:
  def __init__(self):
    self.__head = None
    self.__size = 0

...

  def insert_to_front(self, node):
    if self.__head != None:
      node.set_next(self.get_head())
    self.__head = Node
    self.size += 1

  ...
```

# Linked List Implementation Efficiency

- Accessing head
  - O(1)
- Iterating over elements
  - O(n)
- Memory usage
  - O(n)

# Data Structure Augmentation

- Accessing tail
  - O(n) ☹
  - How can we do better? Easy!

```
class LinkedList:
  def __init__(self):
    self.__head = None
    self.__tail = None
    self.__size = 0


  ...
```

# Common Bugs

Joel's note to self: We will draw the in-memory layout of a linked list (word RAM)

# Week 3 – Sorting & Linear DS

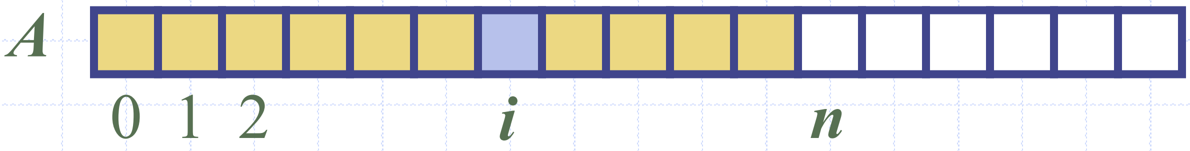
1. Bucket-sort and radix-sort
2. Arrays
3. Linked Lists
4. Extensible Lists and Amortization
5. Comparing Linear Structures

# Example: Sequence of List operations

| Method | Return Value | List Contents |
|---|---|---|
| add(0, A) | – | (A) |
| add(0, B) | – | (B, A) |
| get(1) | A | (B, A) |
| set(2, C) | *error* | (B, A) |
| add(2, C) | – | (B, A, C) |
| add(4, D) | *error* | (B, A, C) |
| remove(1) | A | (B, C) |
| add(1, D) | – | (B, D, C) |
| add(1, E) | – | (B, E, D, C) |
| get(4) | *error* | (B, E, D, C) |
| add(4, F) | – | (B, E, D, C, F) |
| set(2, G) | D | (B, E, G, C, F) |
| get(2) | G | (B, E, G, C, F) |

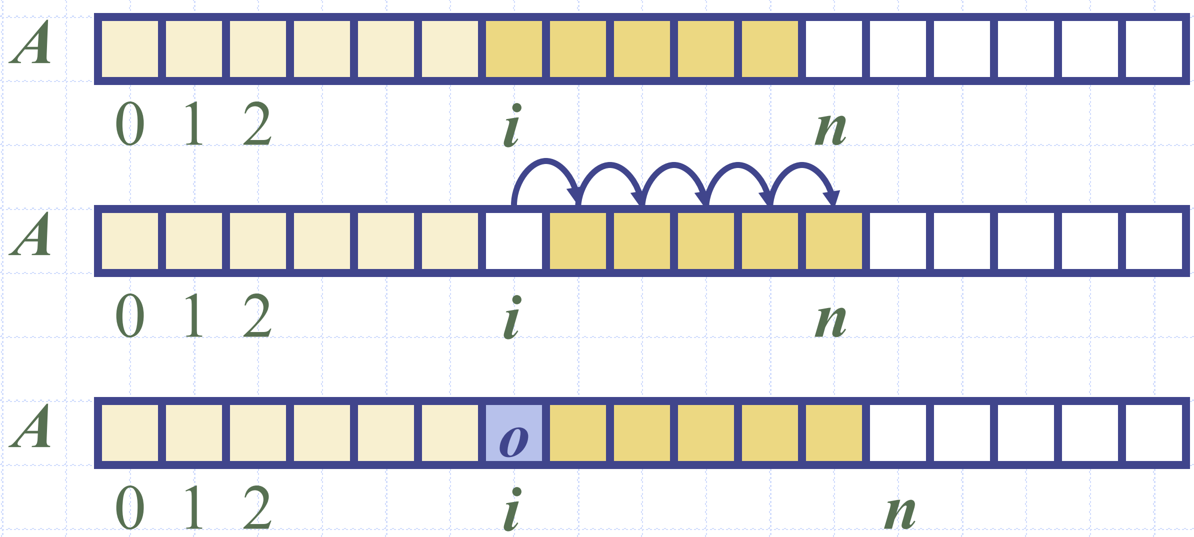
# Array Lists = Extensible Lists = Growable Arrays = Python Lists = …

- An obvious choice for implementing the list ADT is to use an array, **`A`**
- **`get(i)`** and **`set(i,e)`**



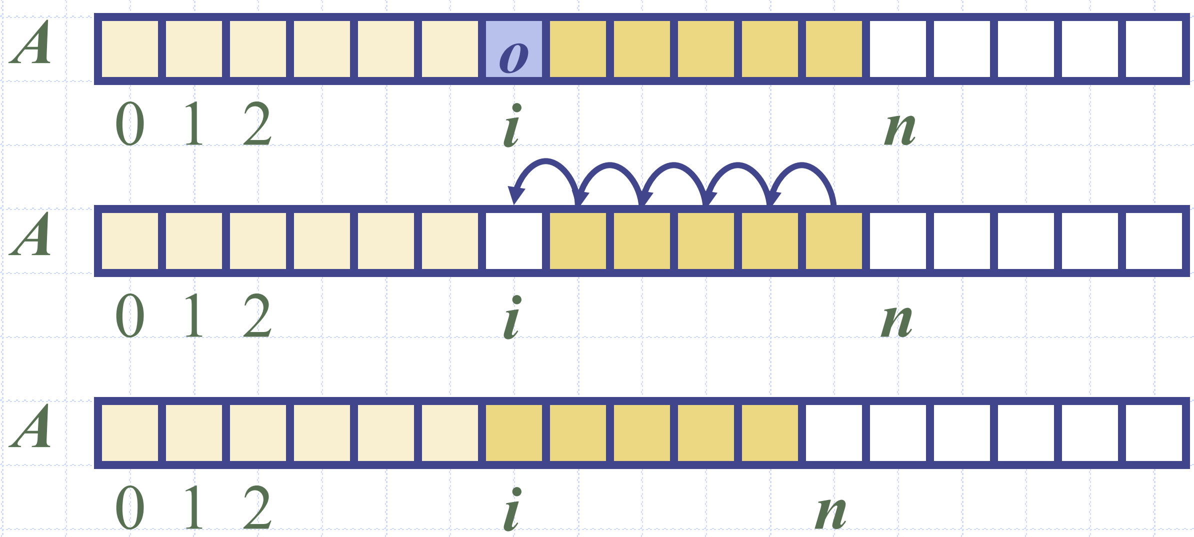
# Insertion

- *add*($i$, $o$) – make room for the new element
- Worst case ($i = 0$), takes $O(n)$ time



# Removal

- $remove(i)$ – fill the hole left by the removed element
- Worst case ($i = 0$), takes $O(n)$ time



# Performance

- Array-based implementation of a dynamic list
  - space used by the data structure is $O(n)$
  - accessing the element at $i$ takes $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time
- ***add*** – when array is full
  - instead of throwing an exception
  - replace the array with a larger one …

# Extensible List

- push(o)/add(o)/append(o): adds element o at the end of the list
- How large should the new array be if we run out of capacity?
  - Incremental strategy
    - increase size by a constant $c$
  - Doubling strategy
    - double the size

# Extensible List

- push(o)/add(o)/append(o): adds element o at the end of the list

```
Algorithm push(o)
  if capacity = S.length – 1 then
    A ← new array of size [something larger]
    capacity = [new larger value]
    for i ← 0 to n–1 do
      A[i] ← S[i]          // copy stuff!
    S ← A                  // update reference to new list
  S[n] ← o
```

# Comparison of Strategies

- Compare incremental and doubling strategies
  - Analysing total time $T(n)$ needed to perform a series of $n$ push operations
- Amortised time of a push operation is the average time taken by a push operation over the series of operations
  - i.e. $T(n) \div n$
  - ***Amortization Intuition: Chocolate…***

# Incremental Strategy Analysis

- Over $n$ push operations, array is replaced $k = n/c$ times, where $c$ is a constant
  - EG: If we extend by c=4 elements each time:
    - After 4 pushes, we extend (n = 4, k = 4/4 = 1)
    - After 4 more pushes, we extend again (n = 8, k = 8/4 = 2)
    - …
    - After n pushes, we have extended **k=n/c** times.

# Incremental Strategy Analysis

- Over $n$ push operations, array is replaced $k = n/c$ times, where $c$ is a constant
- Total time $T(n)$ of a series of $n$ push operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + c(k(k + 1) / 2)$$

- Since $c$ is a constant, $T(n)$ is $O(n + k^2)$, i.e. $O(n^2)$
- Thus, the amortised time of push is $T(n) / n = n^2/\mathbf{n} = O(n)$

# Doubling Strategy Analysis

- Array is replaced $k = \log_2 n$ times
- Total time $T(n)$ of a series of $n$ push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k$$
$$= n + 2^{k+1} - 1 \text{ [geometric series]}$$
$$= n + 2(2^k) - 1$$
$$= n + 2\,(2^{\log n}) - 1$$
$$= 3n - 1$$

- $T(n)$ is $O(n)$
- Amortised time of push is $T(n) \div n = O(1)$

# Further Reading and Up Next

- Data Structures and Algorithms in Python
  - Chapter 5
  - Chapter 7.1 to 7.3
  - Chapter 12.4
- Introduction to Algorithms
  - Chapter 8
  - Chapter 10.1, 10.2