Managing Memory
6.037 – Structure and Interpretation of Computer Programs

Benjamin Barenblat
Massachusetts Institute of Technology

Lecture 8B
Expanding our horizons

So far, we’ve examined just a bit of the overall Scheme story.
So far, we’ve examined just a bit of the overall Scheme story.

primitive data structures (e.g., cons cells)
Expanding our horizons

So far, we’ve examined just a bit of the overall Scheme story.

complex data structures
primitive data structures (e.g., cons cells)
So far, we’ve examined just a bit of the overall Scheme story.

math
↑
lambda calculus
complex data structures
primitive data structures (e.g., cons cells)
RAM
↓
electrical engineering
Implementing cons-cell memory

- We’ve been using the cons-cell abstraction this whole class.

- Computer memory doesn’t really work like that.
Computer Memory

- Conventional memory is an array of locations, each of which has an integer **address**, and stores a single value.

```
  0  1  2  3  4  5  6  7  ... 
```

- Addresses are sequential, so we often move around memory by adding and subtracting values from addresses.
We will model memory using **vectors**.

Also a generally-useful data structure (similar to arrays in other languages).

Vectors support **constant-time** access of an arbitrary element.
(make-vector <size>) → <v>

Returns a vector of the given size.
Vector Operations

- `(make-vector <size>) → <v>`
  - Returns a vector of the given size.

- `(vector-ref <v> <n>) → <elt>`
  - Return the element at index `n` of `v` (0-indexed)

Benjamin Barenblat (MIT)
Vector Operations

- \((\text{make-vector } <\text{size}>\) \rightarrow <\text{v}>\)
  - Returns a vector of the given size.

- \((\text{vector-ref } <\text{v}> <n>) \rightarrow <\text{elt}>\)
  - Return the element at index \(n\) of \(v\) (0-indexed)

- \((\text{vector-set! } <\text{v}> <n> <\text{val}>\) \rightarrow \text{undefined}\)
  - Sets the element at index \(n\) of \(v\).
Vector Operations

- \((\text{make-vector } <\text{size}>)) \rightarrow <v>
  - Returns a vector of the given size.
- \((\text{vector-ref } <v> <n>)) \rightarrow <\text{elt}>
  - Return the element at index \(n\) of \(v\) (0-indexed)
- \((\text{vector-set! } <v> <n> <\text{val}>)) \rightarrow \text{undefined}
  - Sets the element at index \(n\) of \(v\).
- \((\text{vector-length } <v>)) \rightarrow <\text{size}>\)
## Vectors and Lists

<table>
<thead>
<tr>
<th>Lists</th>
<th>Vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant-time append at the beginning</td>
<td>No append at all</td>
</tr>
<tr>
<td>Constant-time insert at any point (with mutation)</td>
<td>No insert at all</td>
</tr>
<tr>
<td>Accessing the $n^{th}$ element takes $O(n)$</td>
<td>Accessing the $n^{th}$ element takes constant time</td>
</tr>
<tr>
<td>Structure can be shared between different lists</td>
<td>Every vector is entirely disjoint</td>
</tr>
<tr>
<td>Rich set of built-in procedures ($\text{map}$, etc.)</td>
<td>Few built-ins (but you can build more)</td>
</tr>
</tbody>
</table>
We will represent **cons cells** using two vectors, **the-cars** and **the-cdrs**.
Representing \texttt{cons} cells

- We will represent \texttt{cons} cells using two vectors, \texttt{the-cars} and \texttt{the-cdrs}.

- A \texttt{cons} cell is an index $i$ into the arrays
  - \texttt{Its car is} $(\text{vector-ref the-cars } i)$
  - \texttt{Its cdr is} $(\text{vector-ref the-cdrs } i)$
We will represent cons cells using two vectors, the-cars and the-cdRs.

A cons cell is an index $i$ into the arrays

- Its car is `(vector-ref the-cars i)`
- Its cdr is `(vector-ref the-cdRs i)`

To represent other data, we’ll use tagging.
We will represent cons cells using two vectors, the-cars and the-cdrs.

A cons cell is an index $i$ into the arrays
- Its car is \(\text{vector-ref the-cars } i\)
- Its cdr is \(\text{vector-ref the-cdrs } i\)

To represent other data, we’ll use tagging.
- $ni$ is a number with value $i$
- $pi$ is a pointer to a pair at index $i$
- $e0$ is the special empty list
An example

((1 2) 3 4)

[Diagram showing a linked list with nodes labeled 1, 2, 3, and 4, connected in the order ((1 2) 3 4).]
An example

((1 2) 3 4)

Index 0 1 2 3 4 5 6 7 8 . . .

the-cars

the-cdrs

Benjamin Barenblat (MIT)
Managing Memory
Lecture 8B
An example

\[((1 \ 2) \ 3 \ 4) \mapsto p1\]

\[
\begin{array}{c}
\text{Index} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \ldots
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{the-cars} & & & & & & & & & \ldots \\
\hline
\text{the-cdrs} & & & & & & & & & \ldots \\
\end{array}
\]
An example

```
((1 2) 3 4) \mapsto p1
```

Index: 0 1 2 3 4 5 6 7 8 ...

| the-cars |   |   |   |   |   |   |   |   | ...
|----------|---|---|---|---|---|---|---|---|------
| the-cdrs |   |   |   |   |   |   |   |   |      ...
```
An example

$$((1 \ 2) \ 3 \ 4) \mapsto p_1$$

```
Index  0  1  2  3  4  5  6  7  8  ...
the-cars  p5  
the-cdrs  
```

Benjamin Barenblat (MIT)  Managing Memory  Lecture 8B  9 / 36
An example

((1 2) 3 4) \mapsto p1

| Index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | ...
|-------|----|----|----|----|----|----|----|----|----|-------
| the-cars | p5 |    |    |    |    |    |    |    |    | ...
| the-cdrs | p2 |    |    |    |    |    |    |    |    | ...  |
An example

$((1\ 2)\ 3\ 4) \mapsto p1$

Index 0 1 2 3 4 5 6 7 8 . . .

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>. . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdtrs</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An example

\[(1 \ 2) \ 3 \ 4) \rightarrow p1\]
An example

$\langle (1 \ 2) \ 3 \ 4 \rangle \rightarrow p1$

\[
\begin{array}{c}
\text{1} & \to & \text{2} & \to & \text{3} & \to & \text{4} \\
\text{5} & \to & & & & & \\
\text{1} & & & & & & \\
\text{2} & & & & & & \\
\text{3} & & & & & & \\
\text{4} & & & & & & \\
\end{array}
\]

Index: 0 1 2 3 4 5 6 7 8 …

<table>
<thead>
<tr>
<th>the-cars</th>
<th>p5</th>
<th>n3</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cdrs</td>
<td>p2</td>
<td>p4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>
An example

((1 2) 3 4) ↦ p1

Index 0 1 2 3 4 5 6 7 8 ...  
the-cars p5 n3  
the-cdrs p2 p4  ...
An example

\[((1, 2), 3, 4) \mapsto p_1\]

Index 0 1 2 3 4 5 6 7 8 ...
the-cars p5 n3 n4 ...
the-cdrs p2 p4 ...

Benjamin Barenblat (MIT)  Managing Memory  Lecture 8B

9 / 36
An example

\(( (1 \ 2) \ 3 \ 4) \mapsto p1 \)
An example

\[((1 \ 2) \ 3 \ 4) \rightarrow p1\]

```
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ...
|-------|---|---|---|---|---|---|---|---|---|-------
| the-cars |   | p5 | n3 | n4 |   |   |   |   |   | ...
| the-cdrs | p2 | p4 | e0 |   |   |   |   |   |   | ...
```
An example

$$(((1, 2), 3, 4)) \mapsto p_1$$

```
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ...
|-------|---|---|---|---|---|---|---|---|---|------
| the-cars | p5 | n3 | n4 | n1 |   |   |   |   |   | ...
| the-cdrs  | p2 | p4 | e0 |   |   |   |   |   |   | ...
```
An example

\[(1\ 2\ 3\ 4) \mapsto p1\]

Index 0 1 2 3 4 5 6 7 8 ... 
| the-cars | p5 | n3 | n4 | n1 |           |           |           | ... |
| the-cdr  | p2 | p4 | e0 | p7 |           |           |           | ... |
An example

\[(1, 2) \rightarrow p1\]

| Index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | ...
|-------|----|----|----|----|----|----|----|----|----|-------|
| the-cars | p5 | n3 | n4 | n1 |     |    |    |    |    | ...
| the-cdrs | p2 | p4 | e0 | p7 |     |    |    |    |    | ...  |
An example

\[(1 \ 2) \ 3 \ 4) \mapsto p1\]

Index 0 1 2 3 4 5 6 7 8 ...

<table>
<thead>
<tr>
<th>the-cars</th>
<th>p5</th>
<th>n3</th>
<th>n4</th>
<th>n1</th>
<th>n2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cdhrs</td>
<td>p2</td>
<td>p4</td>
<td>e0</td>
<td>p7</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
An example

\((1 \ 2) \ 3 \ 4) \mapsto p1\)

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p5</td>
<td>n3</td>
<td>n4</td>
<td>n1</td>
<td>n2</td>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdr</td>
<td>p2</td>
<td>p4</td>
<td>e0</td>
<td>p7</td>
<td>e0</td>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An example

\[(1\ 2)\ 3\ 4) \mapsto p1\]

Index 0 1 2 3 4 5 6 7 8 ...

<table>
<thead>
<tr>
<th>the-cars</th>
<th>p5</th>
<th>n3</th>
<th>n4</th>
<th>n1</th>
<th>n2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cdrs</td>
<td>p2</td>
<td>p4</td>
<td>e0</td>
<td>p7</td>
<td>e0</td>
<td>...</td>
</tr>
</tbody>
</table>
(define (gc-car pair)
  (vector-ref the-cars (pointer-value pair)))

(define (gc-cdr pair)
  (vector-ref the-cdtrs (pointer-value pair)))

(define (gc-set-car! pair new-car)
  (vector-set! the-cars
    (pointer-value pair) new-car))

(define (gc-set-cdr! pair new-cdr)
  (vector-set! the-cdtrs
    (pointer-value pair) new-cdr))
(define (gc-cons car cdr)
    (let ((pair (gc-new-pair)))
        (gc-set-car! pair car)
        (gc-set-cdr! pair car)
        pair))
(define *free* 0)
(define (gc-new-pair)
  (let ((new-pair *free*))
    (set! *free* (+ *free* 1))
    (tag-pointer 'pair new-pair)))
(define *free* 0)
(define (gc-new-pair)
  (let ((new-pair *free*))
    (set! *free* (+ *free* 1))
    (tag-pointer 'pair new-pair)))

What’s wrong?
(define (find-primes n)
  (define (helper ns)
    (cons (car ns) (find-primes
      (filter (lambda (i) (not (divides? i n))) (cdr ns)))
    (helper (cdr (integers-less-than n)))))

  (2 3 4 5 6 7 8 9 10 11 12 ...)
  (3 5 7 9 11 13 15 17 19 21 ...)
  (5 7 11 13 17 19 23 25 27 ...)

Goal: Re-use storage

- Every filter step generates intermediate lists
- But those lists can never be accessed again!
- We can re-use that storage space
The Big Idea

- We can simulate a machine with infinite memory by detecting and re-using memory that can never be used again.
- How do we do that?
Reachability

- There is a set of objects (the “root set”) the program can directly access (e.g. the global environment)
- Objects can point to other objects (e.g. cons cells, the environment pointer of a lambda)
- Any object that is transitively reachable by following pointers from the root set is **live** and must be preserved.
- Anything else is **garbage** and can be reused.
We could keep track of how many pointers there are to each object.

Every time we generate a new reference to an object, we increase the reference count.

- `define`
- `set!`
- `apply` a compound procedure
- ...

Whenever we remove a reference to an object, decrease the count.

- `set!` (The old value)
- After `apply`ing a compound procedure.
- ...

Naïve refcounting leaks circular objects!

```scheme
(define x (list 'a))
(set-cdr! x x)
(set! x 0)
```

Performance impact in many cases.
- Every time you leave a frame, you need to walk its variables
Describe the “root set” explicitly.
- On real hardware, this is the “registers”
- In m-<latex>$\text{eval}$</latex> this is (roughly) the global environment plus the current environment.

Only objects reachable from this set by some sequence of <latex>\text{car}</latex> and <latex>\text{cdr}</latex> can ever matter.

Any memory that is not accessible in this way is garbage, and can be reused.
mark-and-sweep is one of the simplest garbage collection algorithms, composed of two phases:
mark-and-sweep

- **mark-and-sweep** is one of the simplest garbage collection algorithms, composed of two phases:
  1. Starting from the root set, recursively **mark** every reachable object.
mark-and-sweep is one of the simplest garbage collection algorithms, composed of two phases:

1. Starting from the root set, recursively mark every reachable object.
2. Sweep all of memory, collecting every unmarked object into the free list.
mark-and-sweep

- mark-and-sweep is one of the simplest garbage collection algorithms, composed of two phases:
  1. Starting from the root set, recursively mark every reachable object.
  2. Sweep all of memory, collecting every unmarked object into the free list.

- Allocation then takes place by removing new pairs from the free list.
root

1-------2-------4
      |       |
      5-----7-----4

1-------2

Index 0 1 2 3 4 5 6 7 8 ...
the-cars p5 n3 n4 n1 n2 ...
the-cdrs p2 p4 e0 p7 e0 ...
the-marks ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
### Index

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td>p5</td>
<td>n3</td>
<td></td>
<td>n4</td>
<td>n1</td>
<td></td>
<td>n2</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>the-cdrss</td>
<td>p2</td>
<td>p4</td>
<td>e0</td>
<td>p7</td>
<td>e0</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-marks</td>
<td>#t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Index 0 1 2 3 4 5 6 7 8 ...
the-cars p5 n3 n4 n1 n2 ...
the-cdrs p2 p4 e0 p7 e0 ...
the-marks #t #t #t #t #t #t #t #t #t...
The diagram represents a memory management structure with nodes labeled as follows:

- **Root Node**: The topmost node in the diagram.
- **Index 1, 2, 3, 4, 5, 6, 7, 8**: These nodes are connected to the root node and are part of the memory structure.
- **The-Cars**: An array of nodes with values p5, n3, n4, n1, n2, and ellipses indicating continuation.
- **The-Cdr**: An array of nodes with values p2, p4, e0, p7, e0, and ellipses indicating continuation.
- **The-Marks**: An array of nodes with values #t, #t, #t, and an arrow indicating continuation.

The structure suggests a free-list memory management system where each node is connected to the next, forming a chain. The values inside the nodes are likely pointers or addresses used in the memory management process.
The diagram represents a linked list structure with nodes labeled as `root`, `the-cars`, `the-cdrs`, and `the-marks`. The index and values are as follows:

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p5</td>
<td>n3</td>
<td>n4</td>
<td>n1</td>
<td>n2</td>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdrs</td>
<td>p2</td>
<td>p4</td>
<td>e0</td>
<td>p7</td>
<td>e0</td>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-marks</td>
<td>#t</td>
<td>#t</td>
<td>#t</td>
<td>#t</td>
<td>#t</td>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
free-list →

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ...
|-------|---|---|---|---|---|---|---|---|---|------|
| the-cars | p5 | n3 | n4 | n1 | n2 |   |   |   |   | ...
| the-cdrs | p2 | p4 | e0 | p7 | e0 |   |   |   |   | ...
| the-marks | #t | #t | #t | #t | #t |   |   |   |   |   |

↑
free-list →

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p5</td>
<td>n3</td>
<td>n4</td>
<td>n1</td>
<td>n2</td>
<td>p3</td>
<td>p4</td>
<td>e0</td>
<td>p7</td>
<td>e0</td>
</tr>
<tr>
<td>the-cdrs</td>
<td>p2</td>
<td>p4</td>
<td>e0</td>
<td>p7</td>
<td>e0</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-marks</td>
<td>#t</td>
<td>#t</td>
<td>#t</td>
<td>#t</td>
<td>#t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Index 0 1 2 3 4 5 6 7 8 ...
the-cars | p5 | n3 | n4 | n1 | n2 | e0 | ...
the-cdr | p2 | p4 | e0 | p7 | e0 | e0 | ...
the-marks | #t | #t | #t | #t | #t | #t | ...
The diagram illustrates a memory management structure with three main components:

1. **Root Node**:
   - The root node is the topmost node in the structure.
   - It is connected to nodes labeled 1, 2, and 4.
   - These nodes are part of the free-list.

2. **Free-List**:
   - The free-list is a circular list of free memory blocks.
   - It is represented by the nodes labeled 1, 2, 3, and 4, connected in a circular manner.

3. **Index Table**:
   - The index table shows the mapping of indices to memory blocks.
   - It includes columns labeled 'Index', 'the-cars', 'the-cdrs', and 'the-marks'.

Here is a breakdown of the index table:

- **Index**:
  - Represents the index positions in the structure.
  - Indexed from 0 to 8, with ellipsis indicating continuation.

- **the-cars**:
  - Columns labeled p5, n3, n4, n1, n2, e0, and ellipsis.

- **the-cdrs**:
  - Columns labeled p2, p4, e0, p7, e0, e0, and ellipsis.

- **the-marks**:
  - Columns labeled #t, #t, #t, #t, and #t.

The diagram and index table together provide a comprehensive view of memory management and allocation.
The figure illustrates a linked list structure with a root node pointing to several nodes, each containing pointers to other nodes. The free-list is shown at the bottom, indicating available memory slots.

The table below represents the contents of three different lists:

<table>
<thead>
<tr>
<th>Index</th>
<th>the-cars</th>
<th>the-cdrs</th>
<th>the-marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p5</td>
<td>p2</td>
<td>#t</td>
</tr>
<tr>
<td>1</td>
<td>n3</td>
<td>p4</td>
<td>#t</td>
</tr>
<tr>
<td>2</td>
<td>n4</td>
<td>e0</td>
<td>#t</td>
</tr>
<tr>
<td>3</td>
<td>n1</td>
<td>p7</td>
<td>#t</td>
</tr>
<tr>
<td>4</td>
<td>n2</td>
<td>e0</td>
<td>#t</td>
</tr>
<tr>
<td>5</td>
<td>e0</td>
<td>e0</td>
<td>#t</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The free-list contains the following entries:

- | e0 | n3 | n4 | n1 | n2 | e0 | ... |
The diagram shows a memory allocation structure with pointers and values. The structure includes:

- **root**: The topmost pointer in the structure.
- **free-list**: A linked list representing free memory blocks.
- **the-cars**: A list storing values such as `p5`, `n3`, `n4`, `n1`, `e0`, `n2`, `e0`, etc.
- **the-cdrs**: Another list storing values like `p2`, `p4`, `e0`, `p7`, `p8`, `e0`, etc.
- **the-marks**: A list indicating free blocks with `#t`.

The index column represents positions in the structure:

```
Index: 0 1 2 3 4 5 6 7 8 ...
```

The values for the-cars and the-cdrs are as follows:

- **the-cars**: `p5`, `n3`, `n4`, `n1`, `e0`, `n2`, `e0`, ...
- **the-cdrs**: `p2`, `p4`, `e0`, `p7`, `p8`, `e0`, ...

The marks are indicated by `#t` for free blocks.

The diagram visually represents the allocation and free status of memory blocks.
Benjamin Barenblat (MIT)  
Managing Memory  
Lecture 8B  

The diagram illustrates a memory management structure, specifically a free-list, with nodes labeled 'the-cars', 'the-cdrs', and 'the-marks'. The free-list shows the next available memory blocks, with indices 0 to 8. The 'the-cars' list contains values p5, n3, n4, n1, e0, n2, e0, and so on. The 'the-cdrs' list contains p2, p4, e0, p7, p8, e0, and so on. The 'the-marks' list indicates the presence of memory blocks with #t.
Index 0 1 2 3 4 5 6 7 8 ...  
the-cars  p5  n3  e0  n4  n1  e0  n2  e0  ...  
the-cdrs  p2  p4  p6  e0  p7  p8  e0  e0  ...  
the-marks #t #t #t #t #t #t #t #t #t ...  

Benjamin Barenblat (MIT)  Managing Memory  Lecture 8B  21 / 36
Index 0 1 2 3 4 5 6 7 8 ...
\[\begin{array}{c|cccccccc}
\text{the-cars} & p5 & n3 & e0 & n4 & n1 & e0 & n2 & e0 \\
\text{the-cdrs} & p2 & p4 & p6 & e0 & p7 & p8 & e0 & e0 \\
\text{the-marks} & \#t & \#t & \#t & \#t & \#t & \#t & \#t \\
\end{array}\]
The diagram illustrates a memory management structure with the following components:

- **Root**: The starting point of the heap.
- **Free-list**: A list of free memory blocks.
- **The-cars**: A list of car objects, each with a pointer to a cdr.
- **The-cdrs**: A list of cdr objects, each with a pointer to a mark.
- **The-marks**: A list of mark objects, each with a pointer to a cdr.

The diagram shows how pointers are connected between these components, illustrating the flow of memory allocation and deallocation.

### Index Table

<table>
<thead>
<tr>
<th>Index</th>
<th>the-cars</th>
<th>the-cdrs</th>
<th>the-marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p5</td>
<td>p2</td>
<td>#t</td>
</tr>
<tr>
<td>1</td>
<td>n3</td>
<td>p4</td>
<td>#t</td>
</tr>
<tr>
<td>2</td>
<td>e0</td>
<td>p6</td>
<td>#t</td>
</tr>
<tr>
<td>3</td>
<td>n4</td>
<td>e0</td>
<td>#t</td>
</tr>
<tr>
<td>4</td>
<td>n1</td>
<td>p7</td>
<td>#t</td>
</tr>
<tr>
<td>5</td>
<td>e0</td>
<td>p8</td>
<td>#t</td>
</tr>
<tr>
<td>6</td>
<td>n2</td>
<td>e0</td>
<td>#t</td>
</tr>
<tr>
<td>7</td>
<td>e0</td>
<td>e0</td>
<td>#t</td>
</tr>
<tr>
<td>8</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

This table represents the state of memory at a particular point, showing the pointers and data structure relationships.
Index 0 1 2 3 4 5 6 7 8 ...  
the-cars  e0  p5  n3  e0  n4  n1  e0  n2  e0 ...  
the-cdrs  p3  p2  p4  p6  e0  p7  p8  e0  e0 ...  
the-marks  #t  #t  #t  #t  #t  #t  #t  #t ...  

Benjamin Barenblat (MIT)  Managing Memory  Lecture 8B 21 / 36
The diagram shows a linked list structure with nodes labeled as follows:

- **root**: The top node of the structure.
- **free-list**: A list of free nodes, indicated by the arrow pointing to the right.

The structure includes:

- **the-cars**: A list where each node contains a value and a pointer to the next node. The values are:
  - Index 0: e0
  - Index 1: p5
  - Index 2: n3
  - Index 3: e0
  - Index 4: n4
  - Index 5: n1
  - Index 6: e0
  - Index 7: n2
  - Index 8: e0

- **the-cdrs**: A list where each node contains a value and a pointer to the next node. The values are:
  - Index 0: p3
  - Index 1: p2
  - Index 2: p4
  - Index 3: p6
  - Index 4: e0
  - Index 5: p7
  - Index 6: p8
  - Index 7: e0
  - Index 8: e0

- **the-marks**: A list where each node contains a value and a pointer to the next node. The values are:
  - Index 0: #t
  - Index 1: #t
  - Index 2: #t
  - Index 3: #t
  - Index 4: #t
  - Index 5: #t
  - Index 6: #t
  - Index 7: #t
  - Index 8: #t
(define (mark p)
  (if (and (gc-pair? p)
            (not (vector-ref
                 the-marks
                 (pointer-value p))))
      (begin
       (vector-set! the-marks
                    (pointer-value p) #t)
       (mark (gc-car p))
       (mark (gc-cdr p))))))
(define (sweep i)
  (if (not (vector-ref the-marks i))
      (begin
        (vector-set! the-cars i *gc-nil*)
        (vector-set! the-cdrs i *free-list*)
        (set! *free-list* (tag-pointer 'pair i)))
    (if (> i 0)
        (sweep (- i 1)))
  )
(define (mark-and-sweep root)
  (clear-all-marks)
  (mark root)
  (set! *free-list* *gc-nil*)
  (sweep (- *memory-size* 1)))
(define (mark-and-sweep-new-pair)
  (if (eq? *free-list* *gc-nil*)
      (begin (mark-and-sweep root)
              (if (eq? *free-list* *gc-nil*)
                  (error "Out of memory")))
      (let ((pair *free-list*))
          (set! *free-list* (gc-cdr *free-list*)
                  pair)))
mark-and-sweep: problems

- How do we keep track of state during mark?
mark-and-sweep: problems

- How do we keep track of state during mark?
- sweep needs to examine all of memory.
mark-and-sweep: problems

- How do we keep track of state during mark?
- sweep needs to examine all of memory.
- Heap fragmentation becomes a big problem.
An alternate plan: Stop-and-copy

To solve these problems, many real systems use some form of a copying garbage collector.
An alternate plan: Stop-and-copy

- To solve these problems, many real systems use some form of a copying garbage collector.
- Our stop-and-copy collector maintains two regions of memory, the working memory and the free memory.
To solve these problems, many real systems use some form of a **copying** garbage collector.

Our **stop-and-copy** collector maintains two regions of memory, the **working** memory and the **free** memory.

When we run out of memory, we **copy** live objects into the free memory, and switch the roles of the halves.
Stop-and-Copy

- We allocate pairs as we did initially with a *free* pointer.
Stop-and-Copy

- We allocate pairs as we did initially with a *free* pointer.
- When we run out of memory, we switch the free and working memories, and we **relocate** root into the new working memory.
We allocate pairs as we did initially with a *free* pointer.

When we run out of memory, we switch the free and working memories, and we relocate root into the new working memory.

We use a new pointer, scan, initially pointing at the start of the new working memory.

As long as scan < *free*, we relocate the car and cdr of scan, and increment scan.
To relocate a pointer:

- If the value it points to has already been copied, update it to point at the new location.
- Otherwise, allocate a new pair, copy the pair it points to there, and update it.
To relocate a pointer:

- If the value it points to has already been copied, update it to point at the new location.
- Otherwise, allocate a new pair, copy the pair it points to there, and update it.
  - Replace the car of the old pair with a tag known as a broken heart.
  - Replace the cdr of the old pair with the pair’s new address.
root: p1

| Index | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | ...
|-------|---|----|----|----|----|----|----|----|----|-------
| the-cars | p4 | n3 | n1 | n2 | p6 | ... |
| the-cdrss | p7 | e0 | p6 | e0 | p3 | ... |
root: p1

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p4</td>
<td>n3</td>
<td>n1</td>
<td>n2</td>
<td>p6</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdrers</td>
<td>p7</td>
<td>e0</td>
<td>p6</td>
<td>e0</td>
<td>p3</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cdrers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*scan*
root: p1

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p4</td>
<td>n3</td>
<td>n1</td>
<td>n2</td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdtrs</td>
<td>p7</td>
<td>e0</td>
<td>p6</td>
<td>e0</td>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free* ↓

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cdtrs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*scan* ↑
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p4</td>
<td>n3</td>
<td>n1</td>
<td>n2</td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
<tr>
<td>the-cdrs</td>
<td>p7</td>
<td>e0</td>
<td>p6</td>
<td>e0</td>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>

*free* ↓

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
<tr>
<td>new-cdrs</td>
<td>p7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>

*scan* ↑
**root: p0**

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>n3</td>
<td>n1</td>
<td>n2</td>
<td>p6</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdr3s</td>
<td>p0</td>
<td>e0</td>
<td>p6</td>
<td>e0</td>
<td>p3</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p4</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cdr3s</td>
<td>p7</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*scan*
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td></td>
<td></td>
<td>n3</td>
<td>n1</td>
<td>n2</td>
<td>p6</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>the-cdr s</td>
<td>p0</td>
<td>e0</td>
<td>p6</td>
<td>e0</td>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>new-cdr s</td>
<td>p7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*scan*
root: \texttt{p0}

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td>n3</td>
<td>n1</td>
<td>n2</td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>the-cdrs</td>
<td>p0</td>
<td>e0</td>
<td>p6</td>
<td>e0</td>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*free* \quad \downarrow

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>new-cdrs</td>
<td>p7</td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*scan* \quad \uparrow
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td>n3</td>
<td></td>
<td>n2</td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdtrs</td>
<td>p0</td>
<td>e0</td>
<td>p1</td>
<td>e0</td>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cdtrs</td>
<td>p7</td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*scan*
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p0</td>
<td>n3</td>
<td></td>
<td>n2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdrses</td>
<td>e0</td>
<td>p1</td>
<td>e0</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cdres</td>
<td>p2</td>
<td>p6</td>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*scan*
root:  p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td>n3</td>
<td></td>
<td>n2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdrs</td>
<td>p0</td>
<td>e0</td>
<td>p1</td>
<td>e0</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td>p6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cdrs</td>
<td>p2</td>
<td>p6</td>
<td>p3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*scan*
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td>n3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>the-cdrs</td>
<td>p0</td>
<td>e0</td>
<td>p1</td>
<td>p3</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td>p6</td>
<td>n2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>new-cdrs</td>
<td>p2</td>
<td>p3</td>
<td>p3</td>
<td>e0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*scan*
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td></td>
<td>n3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>the-cdrs</td>
<td>p0</td>
<td>e0</td>
<td>p1</td>
<td>p3</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td>p6</td>
<td>n2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>new-cdrs</td>
<td>p2</td>
<td>p3</td>
<td>p3</td>
<td>e0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*scan*
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>p0</td>
<td>e0</td>
<td>p1</td>
<td>p3</td>
<td>p2</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdrs</td>
<td>n3</td>
<td>n1</td>
<td>p3</td>
<td>n2</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td>p3</td>
<td>n2</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cdrs</td>
<td>p2</td>
<td>p3</td>
<td>p3</td>
<td>e0</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*scan*
root: p0

```
Index  0  1  2  3  4  5  6  7  8  ...
the-cars
the-cdrs
```

*free*

```
Index  0  1  2  3  4  5  6  7  8  ...
new-cars
new-cdr
```

*scan*
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>p0</td>
<td>p4</td>
<td>p1</td>
<td>p3</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*free*

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>p1</td>
<td>n1</td>
<td>p3</td>
<td>n2</td>
<td>n3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>new-cdrs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>p2</td>
<td>p3</td>
<td>p4</td>
<td>e0</td>
<td>e0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*scan*
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the-cdhrs</td>
<td>p0</td>
<td>p4</td>
<td>p1</td>
<td>p3</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td>p3</td>
<td>n2</td>
<td>n3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new-cdhrs</td>
<td>p2</td>
<td>p3</td>
<td>p4</td>
<td>e0</td>
<td>e0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free*

*scan*
root: p0

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the-cars</td>
<td>🍓</td>
<td>🍓</td>
<td>🍓</td>
<td>🍓</td>
<td>🍓</td>
<td>🍓</td>
<td>🍓</td>
<td>🍓</td>
<td>🍓</td>
<td>...</td>
</tr>
<tr>
<td>the-cdrs</td>
<td>p0</td>
<td>p4</td>
<td>p1</td>
<td>p3</td>
<td>p2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*free*

↓

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-cars</td>
<td>p1</td>
<td>n1</td>
<td>p3</td>
<td>n2</td>
<td>n3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>new-cdrs</td>
<td>p2</td>
<td>p3</td>
<td>p4</td>
<td>e0</td>
<td>e0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

*scan*
root: p0

\[
\begin{array}{cccccccccc}
\text{Index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \ldots \\
\hline
\text{new-cars} & & & & & & & & & & \\
\text{new-cdrs} & p0 & p4 & p1 & p3 & p2 & & & & & \\
\end{array}
\]

\*free* \quad \downarrow

\[
\begin{array}{cccccccccc}
\text{Index} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \ldots \\
\hline
\text{the-cars} & p1 & n1 & p3 & n2 & n3 & & & & & \\
\text{the-cdrs} & p2 & p3 & p4 & e0 & e0 & & & & & \\
\end{array}
\]

\*scan* \quad \uparrow
(define (stop-and-copy)
  (define (loop scan)
    (if (< scan *free*)
      (begin
        (vector-set! new-cars scan
          (relocate
            (vector-ref new-cars scan)))
        (vector-set! new-cdrs scan
          (relocate
            (vector-ref new-cdrs scan)))
        (loop (+ scan 1)))))
  (set! *free* 0)
  (set! *root* (relocate *root*))
  (loop 0)
  (swap-spaces))
(define (relocate ptr)
  (if (gc-pair? ptr)
      (if (broken-heart? (gc-car pair))
        (gc-cdr pair)
        (let ((new-pair *free*))
          (set! *free* (+ 1 *free*))
          (vector-set! new-cars new-pair (gc-car ptr))
          (vector-set! new-cdr's new-pair (gc-cdr ptr))
          (gc-set-car! ptr *broken-heart*)
          (gc-set-cdr! ptr (tag-pointer 'pair new-pair))
          (tag-pointer 'pair new-pair)))
      ptr))
Properties of stop-and-copy

- Since it moves things around, the garbage collector must know about every pointer into the heap.
- Compacts used memory into a single chunk
  - This means allocation is extremely efficient.
- You only get to use half of your memory.
  - But with mark-and-sweep you potentially needed that for the stack.
- Most modern GCs use something that looks more like stop-and-copy than mark-and-sweep.
Think about the kinds of garbage a program creates.

`find-primes` generated a lot of garbage, but it was very short-lived.

In the adventure game, players, brains and items are created and destroyed, but tend to last a while first.

This turns out to be true in general: A large amount of garbage is destroyed very quickly, whereas garbage that sticks around for a while is likely to stick around more.
Generational GC

- **Big Idea:** Have two (or more!) memory pools.
- Allocate everything into a small one, and scan it every time you do a GC.
- If an object survives a few garbage collections, move it into a larger pool, which is only fully scanned rarely.
- Nearly every real modern GC works roughly this way.