Node* table[TABLESIZE];

void insert(string s) {
    int index = hashfunc(s) % TABLESIZE;
    if notInList(s, table[index]) {
        table[index] = new Node(s, table[index]);
    }
}

bool notInList(string s, Node* p) {
    if (p == NULL) return true;
    return (p->key_ != s) && notInList(p->next_);
}
### Static Allocation

Node* table[TABLESIZE];

```cpp
void insert(string s) {
    int index = hashfunc(s) % TABLESIZE;
    if (!notInList(s, table[index])) {
        table[index] = new Node(s, table[index]);
    }
}

bool notInList(string s, Node* p) {
    if (p == NULL) return true;
    return (p->key_ != s) && notInList(p->next_);
}
```

### Stack Allocation

Node* table[TABLESIZE];

```cpp
void insert(string s) {
    int index = hashfunc(s) % TABLESIZE;
    if (!notInList(s, table[index])) {
        table[index] = new Node(s, table[index]);
    }
}

bool notInList(string s, Node* p) {
    if (p == NULL) return true;
    return (p->key_ != s) && notInList(p->next_);
}
```
Dynamic (Heap) Allocation

Node* table[TABLESIZE];

void insert(string s) {
    int index = hashfunc(s) % TABLESIZE;
    if notInList(s, table[index]) {
        table[index] = new Node(s, table[index]);
    }
}

bool notInList(string s, Node* p) {
    if (p == NULL) return true;
    return (p->key_ != s) && notInList(p->next_);
}

Heap Options

Explicit
   User specifies allocation/deallocation.
   Potential for dangling pointers, memory leaks
   Who’s responsible? ("Liveness is a global property")

Implicit
   User specifies only allocations
   Objects automatically deallocated when known to be safe.
**Vocabulary**

*Object* = data item on the heap (not necessarily OOP).

*Live* = will be needed later.

*Dead* = not live.

*Garbage* = objects that are dead.

A garbage collector detects and deallocates garbage.

Deallocating *all* garbage is undecidable

Thus collectors deallocate a subset of the dead objects

**More Vocabulary**

*Roots* = Pointers into the heap from outside

Found in registers, static memory, and the stack

Data on the heap is said to be *reachable* if it can be accessed by following pointers through the heap, starting with one of the roots; *unreachable* otherwise.

>99% of garbage collectors based on the idea *unreachable objects are garbage.*
Reference Counting

For each object in the heap, keep track of its references.

Update the count when pointers are copied, overwritten, or discarded.

Deallocate any object whose count falls to zero.
Initial Machine State

Pointer Deleted
CS132

Pointer Deleted

registrers

memory graph

stack

...
Pointer Redirected

registers

memory graph

stack

42
255
...

3
4
5
1
2
3
42
1
1
2
2
1
3
3
Pointer Redirected
**Program Resumes Allocating**

<table>
<thead>
<tr>
<th>registers</th>
<th>memory graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>1 3</td>
</tr>
<tr>
<td>255</td>
<td>1 4</td>
</tr>
<tr>
<td>...</td>
<td>1 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stack</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 1</td>
</tr>
<tr>
<td></td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>1 2</td>
</tr>
<tr>
<td></td>
<td>2 2</td>
</tr>
</tbody>
</table>

**Advantages**

- Conceptually simple
- Memory can be re-used immediately
- Can be applied just to difficult pieces of data.
- Relatively few long pauses ("incremental" GC)
  - Time proportional to amount of data freed at once
- Can run finalizers (cleanup actions) immediately.
Disadvantages

Very tricky and error-prone to do by hand

Cost of reference-count updates in space and time.

Space requirement for reference counts.

Cyclic data structures problematic.

Initial State

![Diagram of initial state with stack and registers, memory graph with nodes and links]
CS132

Pointer Redirected

registers

memory graph

stack

...
CS132

Program Continues On

CS132

Pointer Removed
Pointer Removed

registers

memory graph

stack

...
**CS132**

**Pointer Removed**

![Diagram showing registers and memory graph with pointers removed]

**CS132**

**Allocation Resumes**

![Diagram showing registers and memory graph with allocation resuming]
Tracing GC

Allocate until memory is entirely filled
When more space is required, determine the reachable data, and deallocate the rest.

Two classical variants
mark-sweep collectors
copying collectors

Mark-Sweep

Collection occurs in two steps
    Traverse the graph of reachable data, and mark the reachable objects.
    Sequentially examine all the objects in the heap, deallocating the ones not marked.
Initial State

Beginning of Mark Phase
Mark Phase
Mark Phase

registers

42
...

stack

memory

1 3 4 5
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Mark Phase
Mark Phase

At Beginning of Sweep Phase
Sweep Phase

![Diagram of Sweep Phase]

Registers

Stack

Memory

Diagram showing the sweep phase with registers, stack, and memory locations.
Sweep Phase

registers

stack

memory

...
Sweep Phase

 registers

  42
  ...

 stack

  ...

 memory

  0
  3 4 5
  0
  1 1 2 2 3 3
  0

 1 0 0

CS132

Sweep Phase

 registers

  42
  ...

 stack

  ...

 memory

  0
  3 4 5
  0
  1 1 2 2 3 3
  0

 1 0 0
Sweep Phase

registers

memory

stack

CS132
Program Continues Allocating

Before and After Summary
Copying Collector

Divide memory into halves (fromspace and tospace)
Allocate objects in fromspace until full. Then,
  Copy reachable data from the fromspace to the tospace.
  Swap the names "fromspace" and "tospace".
  Resume allocating

Fromspace Full: Start Collection
Copying

registers

42
...

stack

...

memory

3 4 5

1 2 6

1 2 3 3

from

to

Copying

registers

42
...

stack

...

memory

3 4 5

1 2 6

1 2 3 3

from

to
Copying

registers

stack

CS132

Improvements

Compilation

Lexing

Review

Copying

...
Copying

Final State
Program Continues Allocating

Before and After Summary

```plaintext
roots

(contiguous free space)
```
1960: McCarthy's LISP
Heap primarily contained "Cons cells" (pairs)
36-bit IBM 704 → Mark & Sweep.

Within a year, 48-bit port used reference counting
First copying collector 3 years later.

Reference Counting

What could we do about cycles?
Reference Counting

What could we do about the space overhead?
   What if we only want 8-bit reference counts?
   1-bit?

Maintaining reference counts takes time.

Recall the initial hash-table implementation…
Maintaining Counts Takes Time

Suppose we only kept track of pointers from other heap objects?

Reference Counting

Suppose we only counted pointers from other heap objects?
Initial Machine State

registrer

42
255
...

stack

...

memory graph

0 1 2
3 4 5
0 1 0
1 1 2 2 1 3 3
0
1
1
1
1
2
2
6
0
1
3
3
42
0
1
2

Pointer Deleted

registrer

42
255
...

stack

...

memory graph

0 1 2
3 4 5
0 1 0
1 1 2 2 1 3 3
0
1
1
1
1
2
2
6
0
1
3
3
42
0
1
2
Mark-Sweep

What if we don't want mark bits attached to objects?
Can we put the bits somewhere else?
Graph traversal is naturally recursive. There's a small but non-zero overhead to function calls. Usually we don't care, but …

How could we avoid recursive calls?

We're garbage collecting because the heap filled up. But we need a stack (or something) for the graph traversal. Where do we put it? What if we run out of room?
Question

What sort of graph traversal does the Cheney Scan implement?

Does this have any advantages/disadvantages?
Tracing

In a semispace collector, what happens to big objects that are used for a long time?

How might we improve this?

Generational GC

- Problem: long-lived data can be copied repeatedly
  - From fromspace to tospace on every collection

- Generational idea: Data that has survived a long time is presumably "important" and will survive
  - We should try to avoid processing old data during GC

- Separate heaps for young, old data ("generations")
  - Mostly GC only looks at the young data ("minor" GC)
  - Only look at older generation when it finally fills up
  - Can reduce bytes copied by orders of magnitude
    - e.g., 2.3M to 500, or 800M to 25M
Nursery Full

Minor GC
Minor GC

Ready to Continue Allocating
Continue Allocating

Nursery Full Again
Eventually FromSpace Fills Up

CS132

Improved Compilation

Lexing

Review

Eventually FromSpace Fills Up

Major GC

CS132

Improved Compilation

Lexing

Review
Is Young Independent of Old?

- What happens if old data points to young data?
  - That is, what if the fromspace contains pointers into the nursery
  - Such references are called "back-pointers"

- Why are these rare in Haskell?
Example

Wrong Answer
Same Example

Right Answer
Is Young Independent of Old?

What happens if old data points to young data?
- Need to treat backpointers as roots.
- When young objects move, need to update the pointers to them in the older generations.

How do we find these pointers without scanning the older generations (doing as much work as a full GC)?
- The only way to get a backpointer is by modifying (assigning to) an object.
- General idea: "write barrier". Remember information about all (relevant) updates.

Generational Collection

Advantages
- Much shorter pauses
- Often dramatically less copying
  - Sometimes orders of magnitude

Disadvantages
- Have to keep track of assignments
- Can increase memory requirements.
Large Objects

- For large objects, copying even once can be overly expensive.
- And larger objects are likely to be long-lived.
- Some generational collectors contain a special heap for large objects.
- Anything large gets automatically allocated there.
- May be managed differently (e.g., mark-sweep).

More Variations

- Incremental collectors
  - Do small amounts of collection, more frequently.
  - Real-time: hard bound on amount of work done by the collector in each step.
    - Increases total collection time, decreases maximum pause.
- Parallel collectors
  - Several processors cooperating together on collection.
- Concurrent collectors
  - Main program and GC run simultaneously.
- Hybrid collectors
  - Variants that combine copying/mark-sweep/ref counts.
- Distributed collectors
  - Heap is divided among several computers.
Which Collector is Best?

Nobody knows; performance depends on
- Client program ("mutator") behavior
  - Frequency of allocation
  - Sizes of objects
    - Number of reads / writes / pointer copies / etc.
- Cache and virtual memory performance
- Order of memory traversal (DFS, BFS, ...)
- When garbage collections occur

Lots of research work collectors
- Incremental, Parallel, Concurrent, Real-Time, …

Another Approach: Regions

Idea:
- Allocate into regions
  - Each region is a heap that never deallocates
  - Which region to use may be decided by the compiler.
  - Deallocate each region as a whole when done with it.
    - (Often, assume a stack of regions.)

Advantages:
- Very efficient allocation and deallocation.
  - Can often detect values that are accessible but dead.

Disadvantages:
- Precise analysis can be tricky
  - (in worst case, everything in 1 region)
Question

Assume we need to do a GC when

%eax = 0x01345568
%ebx = 0x01345560
%ecx = 0x00000003
%edx = 0xFFFFFFFF
%esi = 0x01345784
%esi = 0xDEADBEEF

What are the roots?

Assume a root points to address X in memory, where:

X[-8] = 0x01345568
X[-4] = 0x01345560
X[0] = 0x00000003
X[4] = 0xFFFFFFFF
X[8] = 0x01345784
X[12] = 0xDEADBEEF

Which other objects are reachable?

Finding Roots

- **Conservative Collection**
- **Guess**

- **Tag bits**
  - Take a bit (or two) from every machine word to denote pointer/non-pointer.

- **Lookup tables**
  - At compile time, generate information about contents of stack and registers for each point in program where GC could occur.
Bread crumbs
- Tag heap objects with their size
- Tag objects with pointer locations
- Or just guess, or assume tag bits

Types
- From the types of pointers, deduce the object's layout, and recursively the type and layout of every object it points to.
- Much harder with polymorphism or abstract types