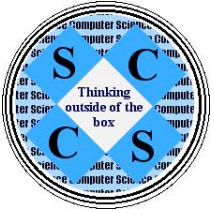




# Pointers

---

- A pointer is a type, where the values are memory addresses and a special null or nil value.
- Pointers are dynamically allocated from a heap
- Heap-dynamic variables are variables dynamically allocated from the heap storage
- Design Decisions
  - **Pointers can be restricted to indirect addressing (aliasing) vs dynamic memory management**
  - **The scope and lifetime of heap variables must be decided upon**
  - **Pointers, References, or both?**



# Pointer Operations

---

- Setting the pointer variable to an address

- C++

```
int global;
int main() {
    int* myPtr;
    int value;
    myPtr = &value; // sets to stack-dynamic address
    myPtr = new int; // sets to heap address
    myPtr = &global; // sets to static-address
    ...
    return 0;
}
```

- The language needs to support an “address of” operator
  - C++ supplies the &



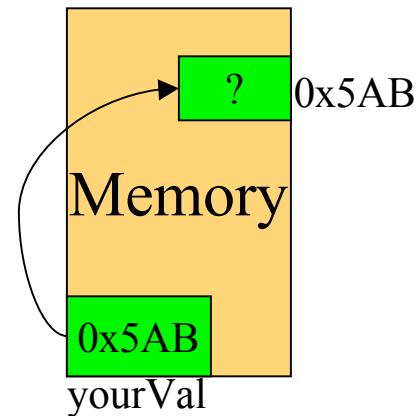
# Pointer Operations Continued

- Referencing a pointer variable in an expression

- Given the following fragment

```
float* value;
```

```
yourVal = 2 + value/5.0;
```



- In C++, 'value' is interpreted as a variable containing a memory location
  - The address stored in value is divided by 5.0
- What's another interpretation?



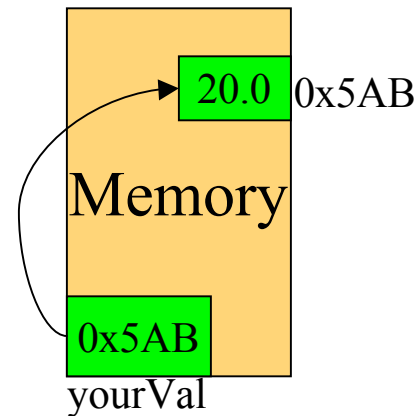
# Pointer Operations: References

- Another interpretation is that 'value' refers to the float it is pointing to

- Given the following fragment

```
float* value;
```

```
yourVal = 2 + value/5.0;
```



- In languages like Java and Ada, 'value' would refer to 20.0
  - The value is divided by 5.0
- In this case, 'value' is **implicitly** treated as a reference pointer



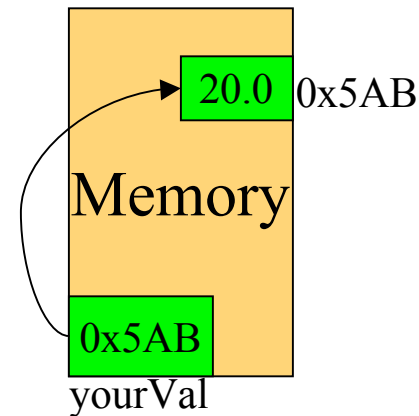
# Pointer Operations: Dereferencing

- Some languages allow the programmer to explicitly dereference a pointer variable

- Given the following fragment

```
float* value;
```

```
yourVal = 2 + *value/5.0;
```



- In C++, ‘\*value’ would refer to 20.0
  - The value is divided by 5.0
- In this case, ‘value’ is dereferenced
- References from the previous slide are automatically dereferenced



# Summary Pointer Operations

---

- The language supplies a way to set the pointer variable or reference
- A pointer variable represents a memory location and requires explicit dereferencing
- A reference variable represents a reference or alias to a value, and utilizes implicit dereferencing



# Problems with Pointers

---

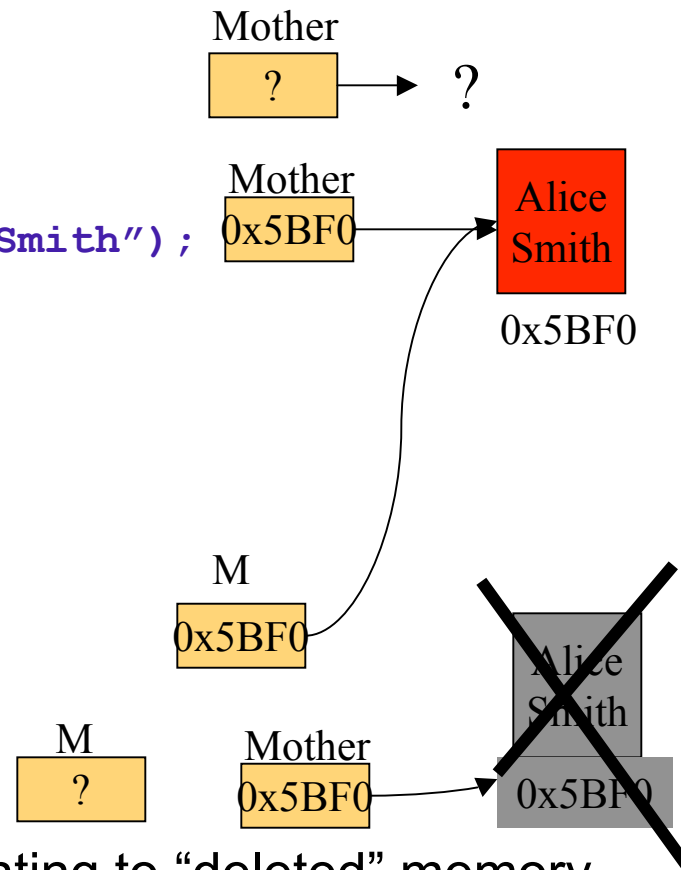
- Dangling pointer or dangling reference
  - **Pointing to invalid memory locations**
- Lost Heap-Dynamic Variable
  - **Memory Leaks**



# Dangling Pointers

- Consider the following scenario

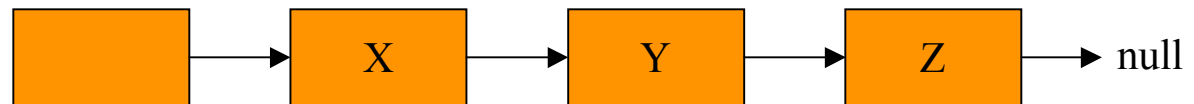
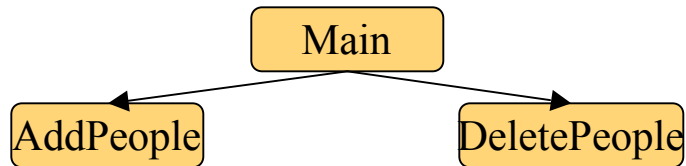
```
int main() {  
    Person* Mother;  
  
    ...  
    Mother = new Person("Alice", "Smith");  
  
    ...  
    AddToList(Mother);  
  
    ...  
    cout << Mother << endl;  
}  
  
...  
void AddToList(Person* M) {  
    ...  
    delete M;  
}
```



- The pointer `Mother` is dangling, pointing to “deleted” memory location



# Memory Leaks

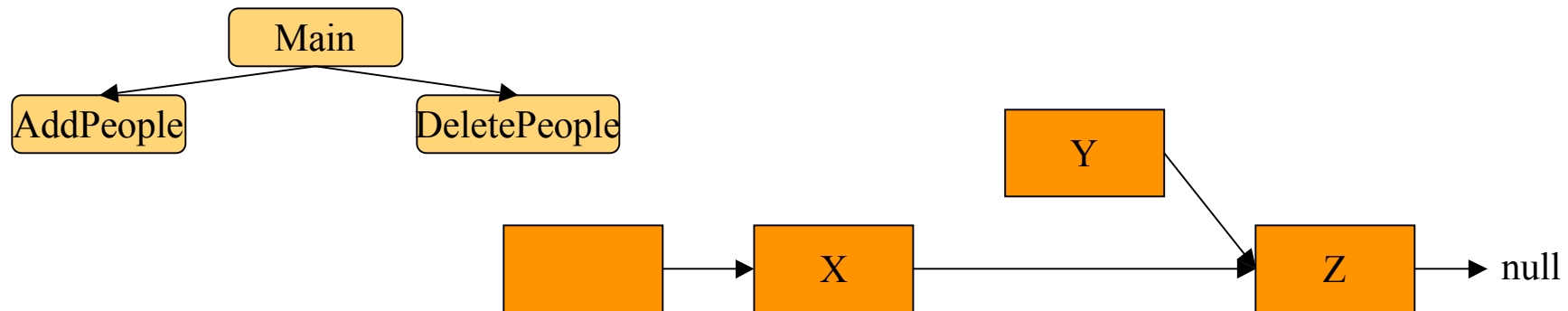


Goal: Delete Y Person

- The program allocates nodes for the list data structure
  - **AddPeople**
- Nodes are removed in DeletePeople



# Memory Leaks

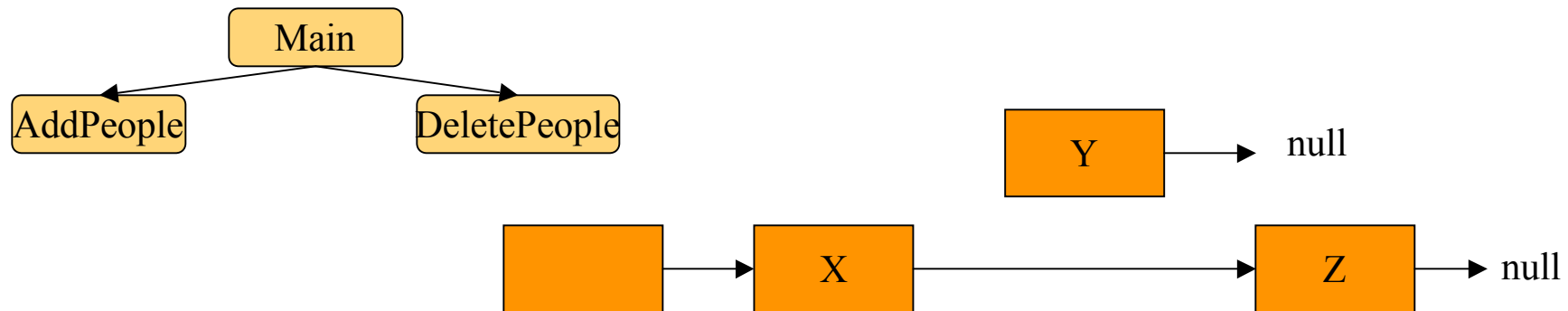


1. Make X point to Z
2. Make Y point to null

- Suppose, the removal of nodes does not “delete” the nodes
  - The memory is never deallocated
  - No pointers to the “removed” nodes

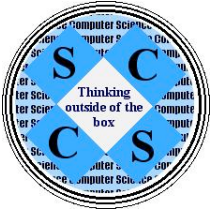


# Memory Leaks



1. Make X point to Z
2. Make Y point to null

- Suppose, the removal of nodes does not “delete” the nodes
  - The memory is never deallocated
  - No pointers to the “removed” nodes
- Non-deallocated garbage nodes can not be reused by the system
  - The system appears to be losing memory (leaking)



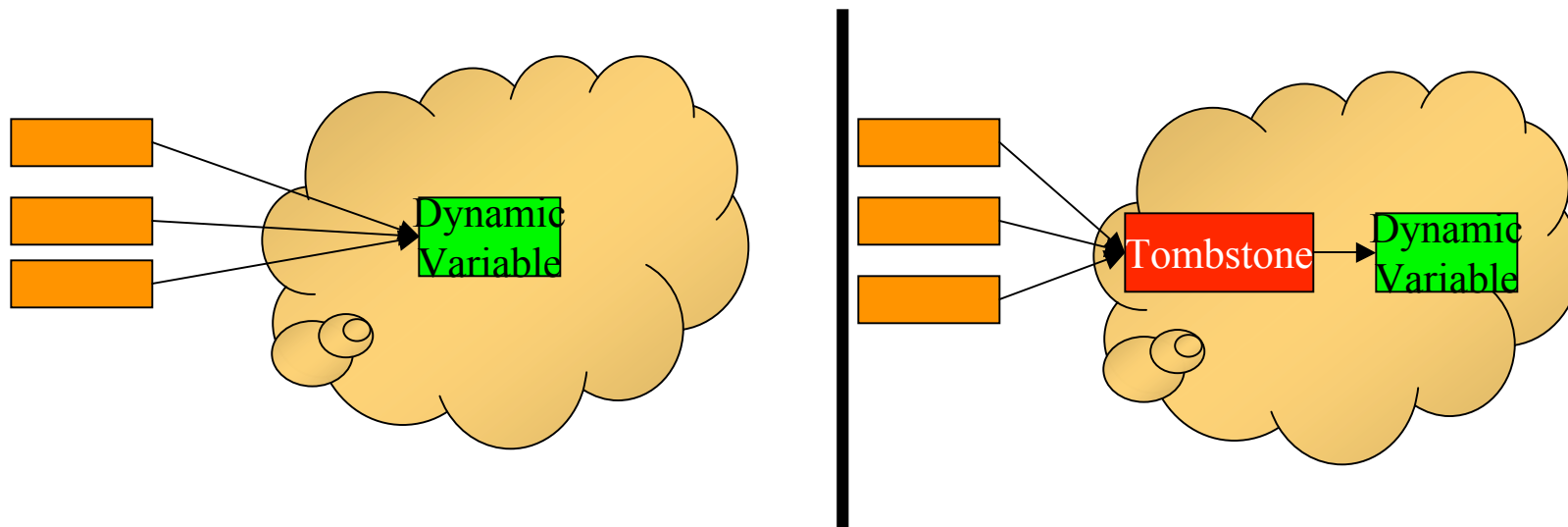
# Language: Solutions to Dangling Pointers/memory leaks

---

- Explicit allocation and deallocation
  - Dangling pointers, memory leaks
  - Very flexible
    - C uses malloc and free
    - C++ uses new and delete
    - Pascal uses new and dispose
- Explicit allocation and implicit deallocation (Garbage Collection)
  - Dangling pointers
  - No memory leaks & loss of flexibility
    - Ada
    - Java
- Implicit allocation and deallocation
  - No Dangling pointers or memory leaks
  - Less flexibility
    - Scheme & Lisp
- Use of references and implicit deallocation
  - No pointers! No memory leaks
  - Flexibility regained?
    - Java



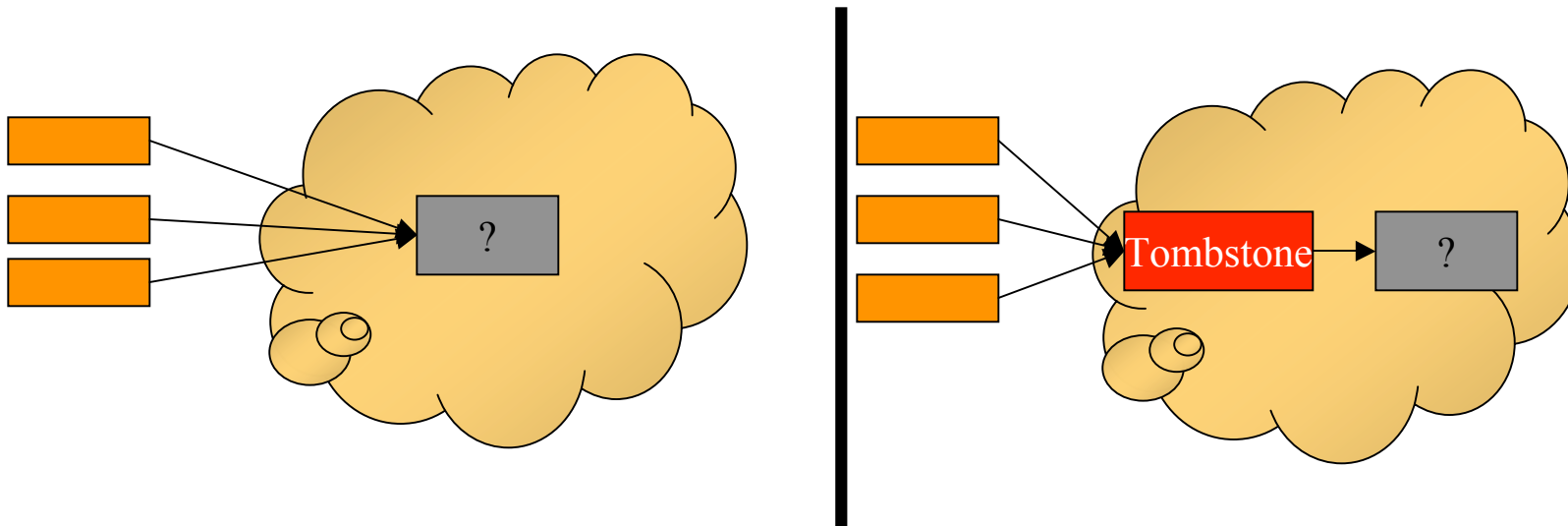
# System-Level: Solving Dangling Pointer Problem



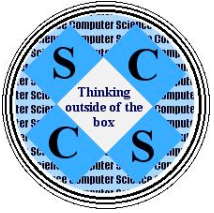
- **Tombstones** can be used to solve this problem
- Tombstones are special heap cells that represent proxies for allocated heap cells.



# System-Level: Solving Dangling Pointer Problem



- When variable is deallocated
  - Without tombstones, leaves dangling pointers
  - With the tombstone, the tombstone can act as a NULL pointer
- Tombstones require additional memory allocations



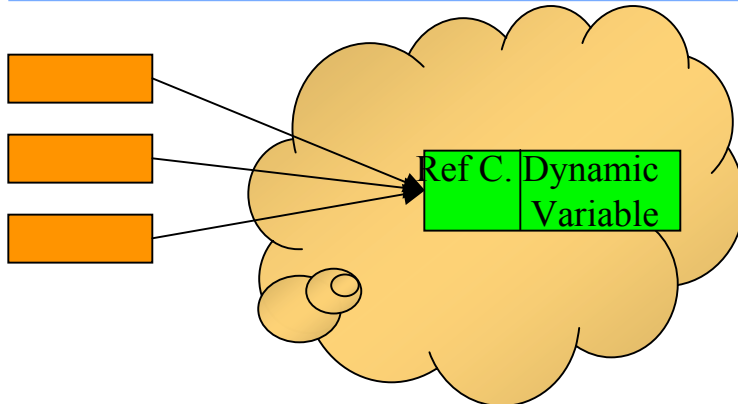
# Heap Management

---

- Heap management requires handling allocation and deallocation strategies
- Heaps can be composed of fixed-sized cells or variable-sized cells
- A cell is analogous to a linked-list node
  - **Ptr to the next cell**
  - **Data area for storing program information**
- Heap has a list of **available** cells (unallocated cells)
- When allocation is required, the Heap removes cells from the available list
- When deallocation is required, there is more to consider
  - **Are there other pointers referencing that memory location?**
  - **If implicit deallocation, then when is it time to deallocate?(Scope of lifetime)**

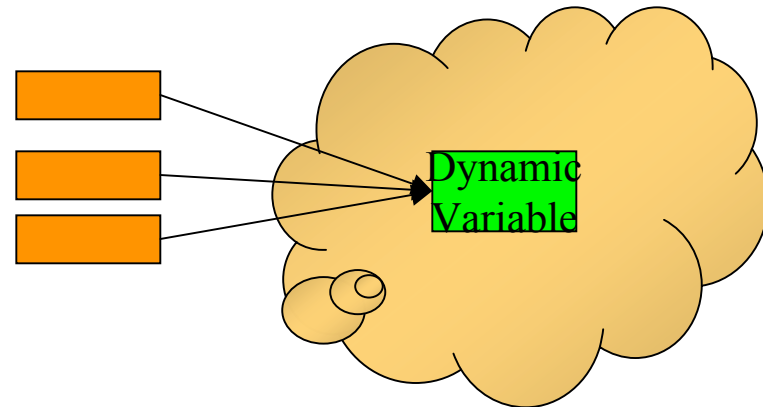


# Heap Management: Deallocation



## Eager Deallocation

- Reference Counters
  - Counter increment on every pointer reference
  - Counter decremented on every pointer deallocation
    - If Ref. C is == 0 then deallocate Dynamic Variable
- Expensive

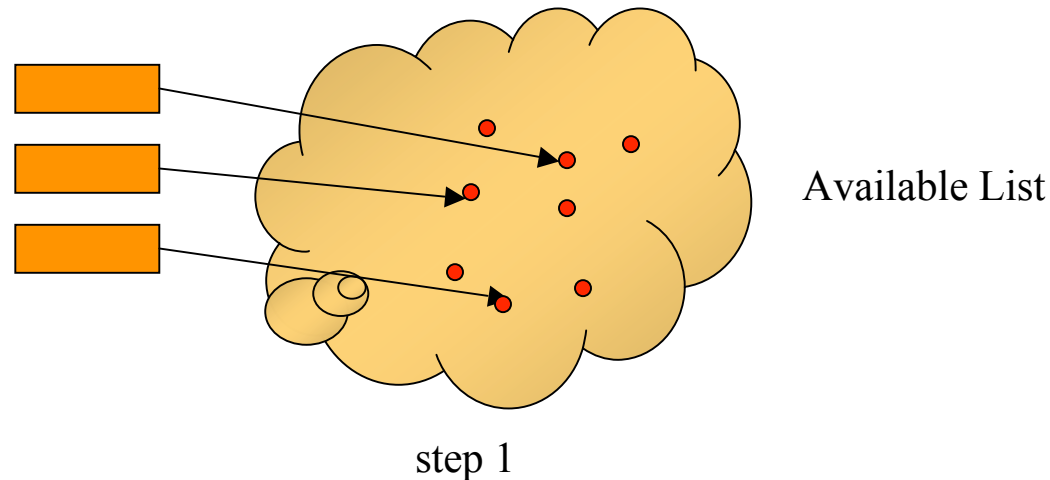


## Lazy Deallocation

- Garbage collection
  - Allocate cells and disconnect (not deallocate) until no free space
  - Process the heap looking for Garbage (cells not referenced)
    - Garbage Collection
- Very slow when most cells are useful
  - Most important time



# Heap Management: Garbage Collection

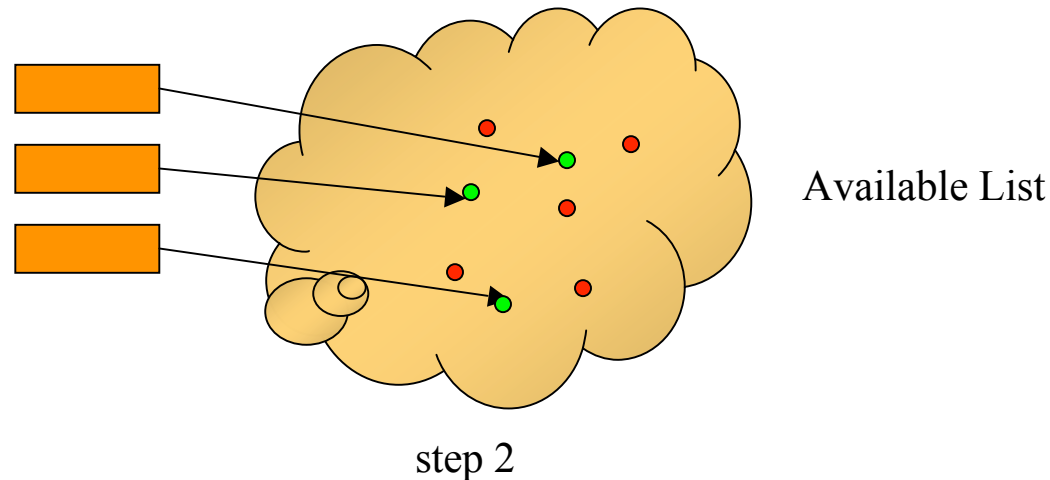


- Algorithm assumes all cells have a bit-field to denote Garbage or not
  1. **Traverse all cells in the heap and mark them as garbage**
  2. Traverse all pointers into the heap
    - a. Any cells that is the destination of a pointer, mark as non-garbage

NOTE: cycles are possible, and must be recognized
  3. Traverse all cells in the heap, and those marked as garbage are reclaimed to the available list



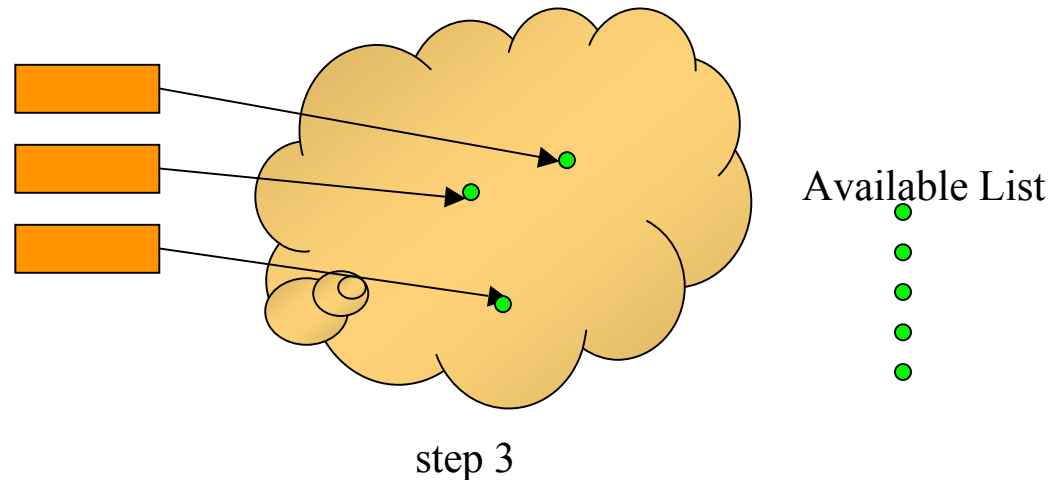
# Heap Management: Garbage Collection



- Algorithm assumes all cells have a bit-field to denote Garbage or not
  1. Traverse all cells in the heap and mark them as garbage
  2. Traverse all pointers into the heap
    - a. Any cells that is the destination of a pointer, mark as non-garbage**NOTE: cycles are possible, and must be recognized**
  3. Traverse all cells in the heap, and those marked as garbage are reclaimed to the available list



# Heap Management: Garbage Collection



- Algorithm assumes all cells have a bit-field to denote Garbage or not
  1. Traverse all cells in the heap and mark them as garbage
  2. Traverse all pointers into the heap
    - a. Any cells that is the destination of a pointer, mark as non-garbageNOTE: cycles are possible, and must be recognized
  3. Traverse all cells in the heap, and those marked as garbage are reclaimed to the available list