CS 1114:
Data Structures – memory allocation

Prof. Graeme Bailey

http://cs1114.cs.cornell.edu

(notes modified from Noah Snavely, Spring 2009)

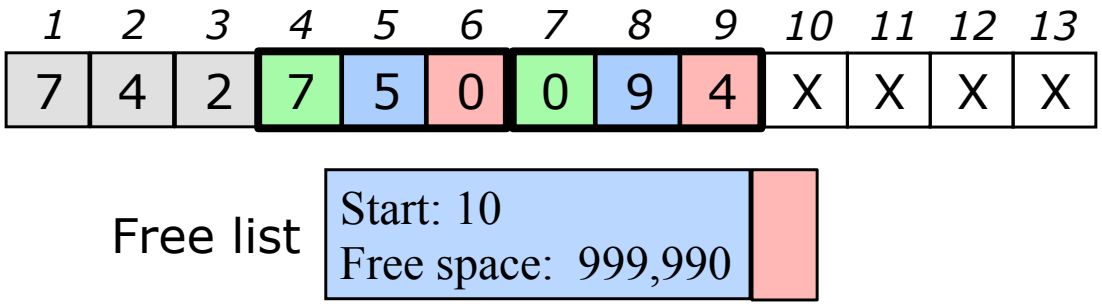
Cornell University
Computer Science

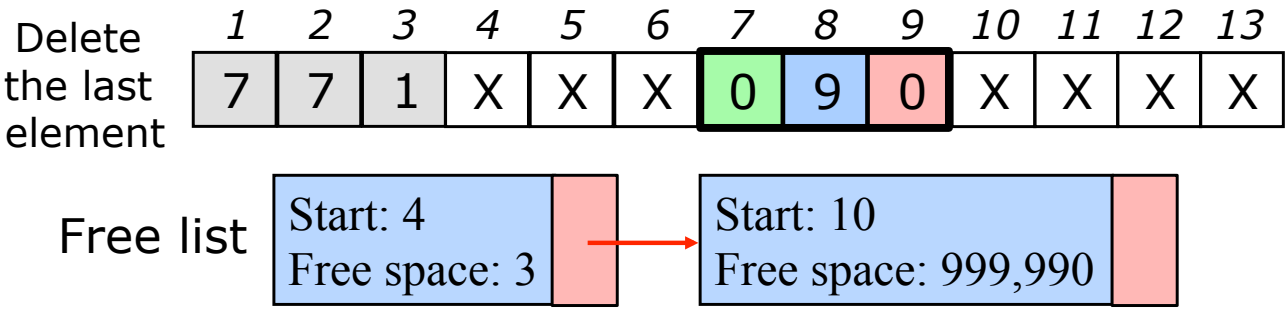
Memory allocation

- Strategy 1: Computer keep tracks of free space at the end

- Strategy 2: Computer keeps a linked list of free storage blocks ("freelist")
  - For each block, stores the size and location
  - When we ask for more space, the computer finds a big enough block in the freelist
  - What if it doesn't find one?

Maintaining a freelist





Allocation issues

- Surprisingly important question:
  - Which block do you supply?
  - The smallest one that the users request fits into?
  - A larger one, in case the user wants to grow the array?

Memory deallocation

- How do we give the computer back a block we're finished with?

- Someone has to figure out that certain values will never be used ever ("garbage"), and should be put back on the free list

  - If this is too conservative, your program will use more and more memory ("memory leak")
  - If it's too aggressive, your program will crash ("blue screen of death")

Memory deallocation

- Two basic options:

1. Manual storage reclamation
   - Programmer has to explicitly free garbage
   - Languages: C, C++, assembler

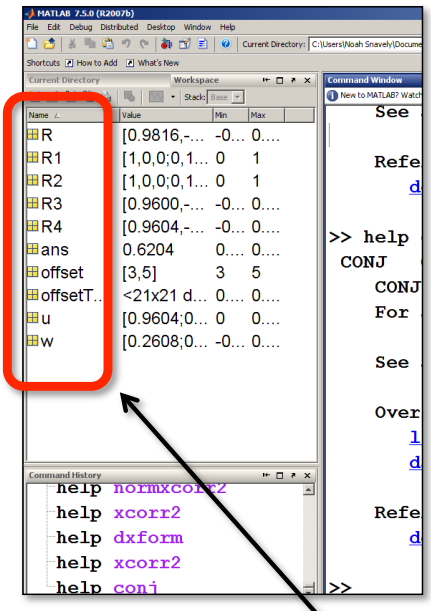
2. Automatic storage reclamation
   - Computer will notice that you're no longer using cells, and recycle them for you
   - Languages: Matlab, Java, C#, Scheme

Manual storage reclamation

- Programmers always ask for a block of memory of a certain size
  - In C, explicitly declare when it is free

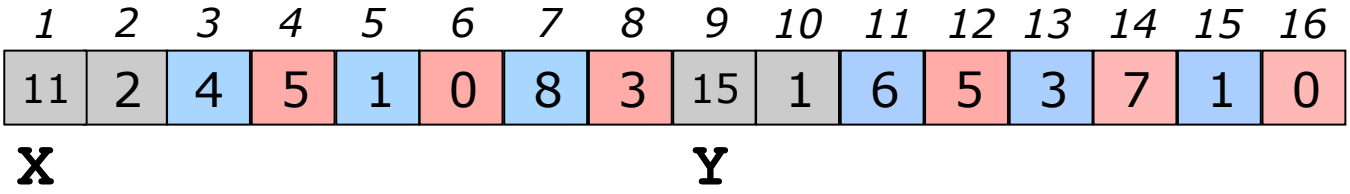
- Desirable but complex invariants:
  1. Everything should be freed when it is no longer going to be used
  2. If we free something, we shouldn't try to use it again
  3. And, it should be freed exactly once!
  4. Minimize fragmentation

Automatic storage reclamation



- "Garbage collection"
- 1st challenge: find memory locations that are still in use by the programmer ("live")
  1. Anything that has a name the programmer can get to (the "root set")
  2. Anything pointed to by a live object

Garbage collection

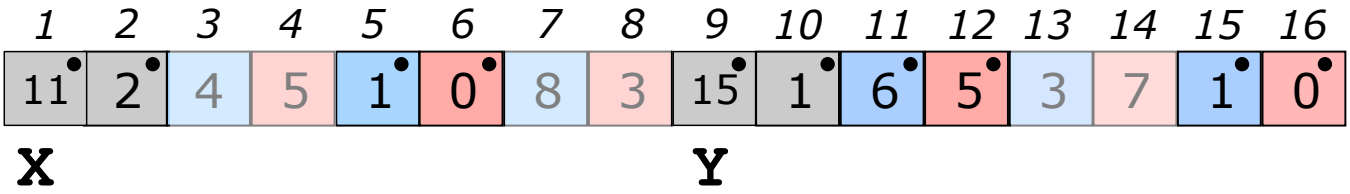


- Two lists, X and Y
- Which cells are live?
- Which cells are garbage?

Simple algorithm: mark-sweep

- Mark: Chase the pointers from the root set, marking everything as you go
- Sweep: Scan all of memory – everything not marked is garbage, and can go back on the free list

Mark and sweep



Root set: { X, Y }

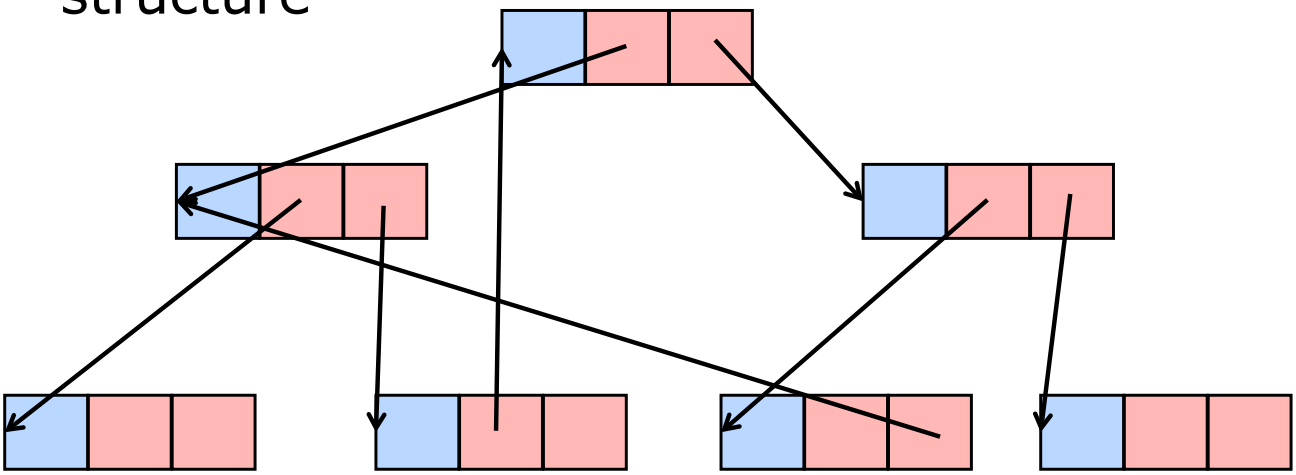
- Mark phase
- Sweep phase

Mark and sweep

- The machine needs to be able to tell where the pointers are (we'll assume that it's up to the programmer to do that)
  - For instance, the programmer will say that the second entry in a cell is a pointer (for singly-linked list)
  - Or, for a doubly-linked list, the first and third entries in a cell are pointers

Mark and sweep

- In general, pointers may have a complex structure



How do we mark, in the general case?

When to do garbage collection?

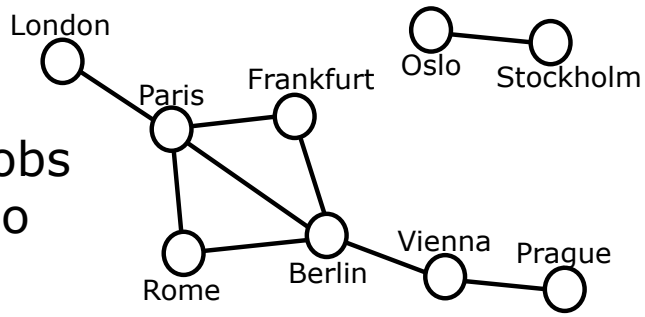
- Option 1 ("stop the world"): Once memory is full, stop everything and run garbage collection
  - Your program will freeze while the garbage is being collected
  - Not good if you're coding the safety monitoring system for a nuclear reactor
- Option 2 ("incremental GC"): Collect garbage in parallel with the main program
  - Needs to be careful not to step on the program

Linked lists – running time

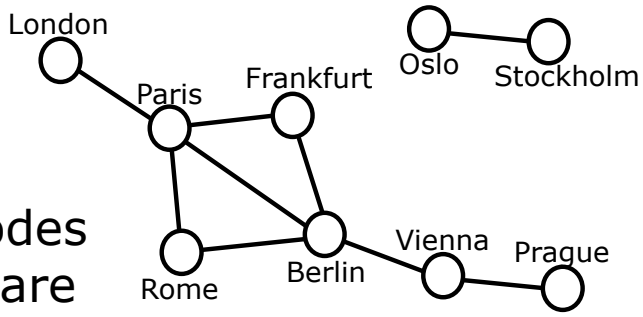
- What about inserting / deleting from somewhere near the *middle* of the list?
  - How can we find items in a list faster than simply starting at one or other end?
- How can we fix this?
- You'll have to wait a bit to learn about this……. ☺

Where are we in the story?

- We started with images
  - Then sought blobs
    - Then decided that blobs are connected, so
- Built graphs
  - Then wanted to find the connected components, so
    - Looked at implementing dfs and bfs, so
- Looked at implementing stacks and queues
  - Then found that arrays were inefficient, so
    - Looked at linked lists
      - Then looked at implementing linked lists
- Arrays were messy, could we use a graph?



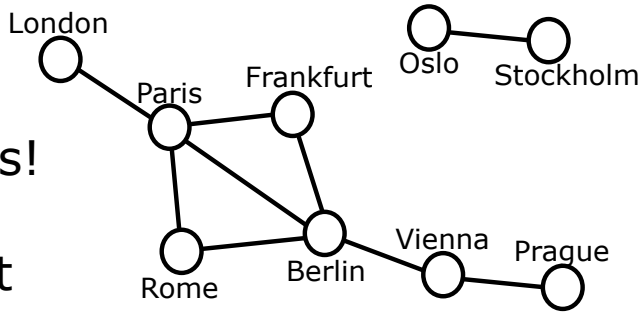
Representing a graph



- Build a big 2-D array
  - Mentally label the rows and columns by the nodes
  - Then insert a 0 if there are no edges between those nodes, or a 1 if they are connected

- What does this array look like if
  - there are lots of edges?
  - there are very few edges?
  - (we call these matrices 'sparse')

Representing a graph



- Can we do better?

- Build an array of linked lists!
  - One linked list per node
  - The edges live in the list

- Details:
  - For each node A, put the nodes which are connected to that node into A's linked list
    - no longer 'sparse'
    - and space isn't being wasted!
    - can represent directed graphs easily