CS5112: Algorithms and Data Structures for Applications

Data representations and real-world efficiency

Ramin Zabih

Sources: Wikipedia; Kevin Wayne, Kleinberg/Tardos

Administrivia

- Prelim (midterm) date: Wednesday October 16
  - In class, closed book
  - Review session in class on October 7
- HW1 is out, due by 10/7
- Lectures will be recorded “Real Soon Now”
- Minor RDZ error in lecture Wednesday

Lecture Outline

- Memory and data representations
- Memory allocation
- The memory hierarchy
- Practical implications for data structure/algorithm design

Computers and arrays

- Notion of “constant time” operations relies on a hardware model
- Computer memory is a large array
  - We will call it M, elements M[0], M[1], etc. Integers, for simplicity.
- In constant time, a computer can:
  - Read any element of M
  - Change any element of M to another element
  - Perform any simple arithmetic operation
- This is more or less what the hardware manual for an x86 describes

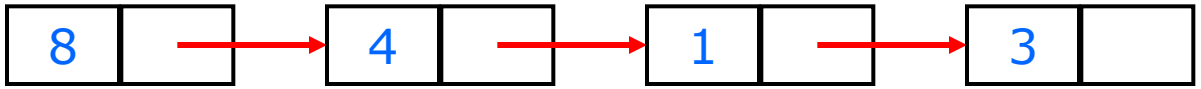
Data structures in memory

- Today we will focus on implementing various data structures using the array M
- We have a contract (API) we wish to fulfill
  - Example: stack contract
    - If we push X onto S and then pop S, we get back X, and S is as before
- We want to fulfill this contract using the memory array M
- We'll look at related data structures also

Linked lists

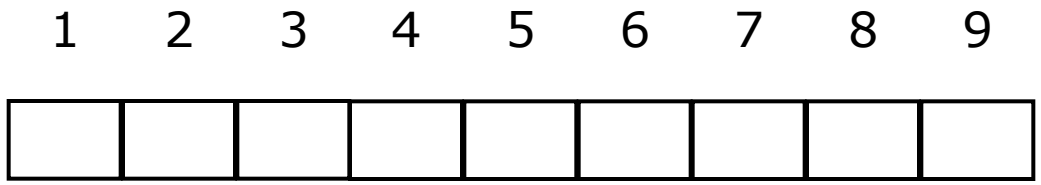
- You can add items to it, and you can process the items you added
- Unlike a stack, don’t need to pop it in order to get at all the items
- Adding items takes constant time
- Getting each item takes linear time
  - Example: searching the list for an element
- You can also delete an element
  - This is **always** the hard part of any data structure, and source of most bugs
  - If you don’t believe me, look up delete on red-black trees

Linked lists



Linked lists as memory arrays

- We'll implement linked lists using M
  - This is very close to what the hardware does

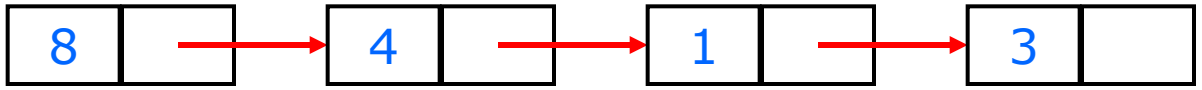


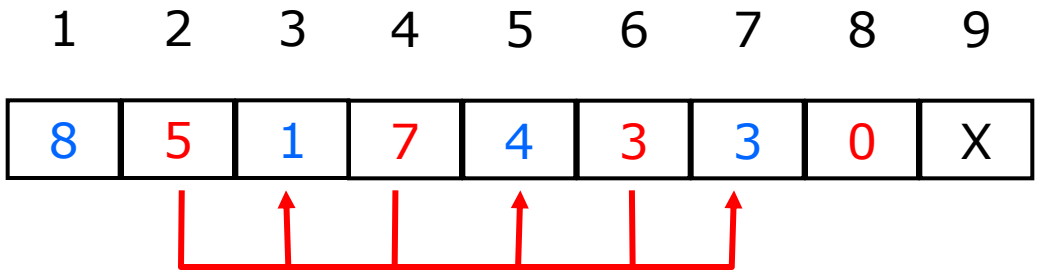
- A linked list contains "cells"
- A value, and where the next cell is
  - We will represent cells by a pair of adjacent array entries

A few details

- I will draw odd numbered entries in blue and even ones in red
  - Odd entries are values
    - Number interpreted as list elements
  - Even ones are “cells”
    - Number interpreted as index of the next cell
    - AKA *location*, *address*, or ***pointer***
- The first cell is M(1) and M(2), for now
- The last cell has 0, i.e. pointer to M(0)
  - Also called a “null pointer”

Example





Access every item in a list

- Start at the first cell, M(1) and M(2)
- Access the first value, M(1)
- The next cell is at location v=M(2)
- If v = 0, we are done
- Otherwise, access the next value, M(v)
- The next cell is at location v’ = M(v+1)
- Keep on going until the next cell is at location k, such that M(k+1) = 0

Adding a header

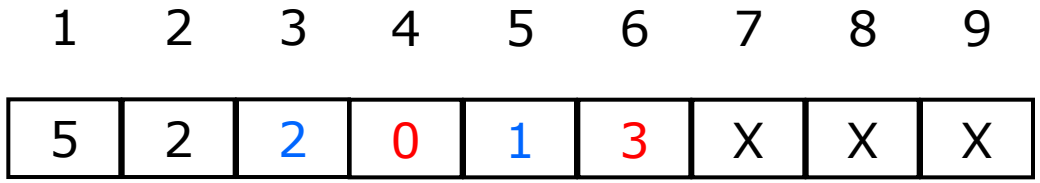
- We can represent the linked list just by the initial cell, but this is problematic
  - Think about deleting the last cell!
  - More subtly, it's also a problem for insertion
- Instead, we will have a few entries in M that are not cells, but instead hold some information about the list
  - Specifically, a pointer to the first element, we use M[1]
  - Having a counter is also useful at times, we use M[2]

Insert and delete

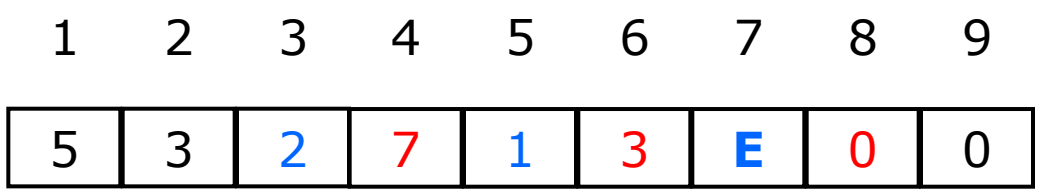
- To insert an element, create a new cell and splice it into the list
  - Various places it could go
  - Need to update header info, and the cell (if any) before the new one
- To delete an element we simply need to update the header and to change some pointers
  - Basically, need to “skip over” deleted element

Linked list with header

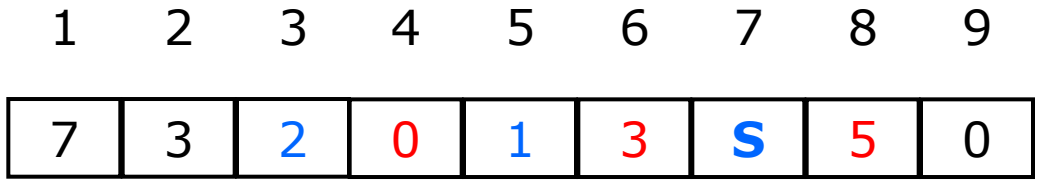
- Initial list:



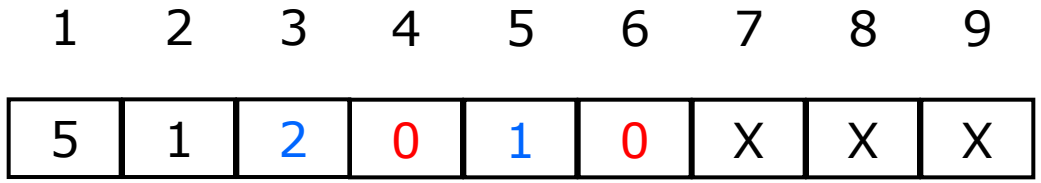
- Insert (end):



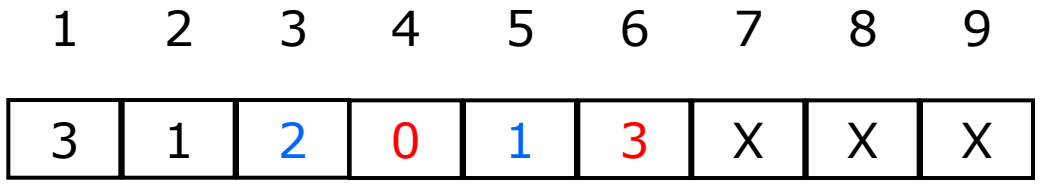
- Insert (start):



- Delete (end):



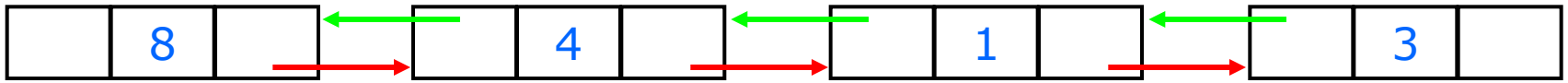
- Delete (start):



Linked lists and deques

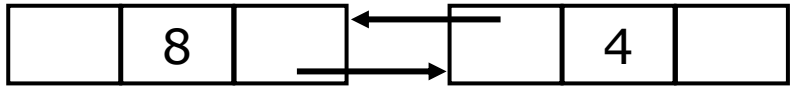
- Linked lists support insertion/deletion at beginning or end
  - What is the complexity of these 4 operations?
  - Is this a good way to implement queues?
- How do we modify linked lists so that we can delete from the end in constant time?
  - We clearly need a pointer to the last element in the header
    - But this is not enough!

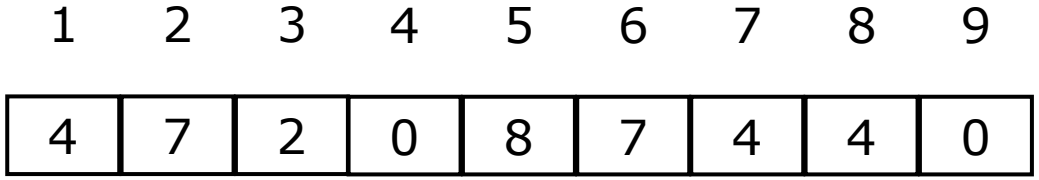
Doubly linked lists



A doubly-linked list in memory

- M[1] points to start
- M[2] points to end
- M[3] is number of cells
- Cells have 3 elements
  - Before, value, after
  - Before & after are pointers
    - Can be null





Notes on doubly-linked lists

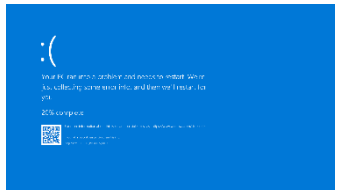
- Inserting and deleting is fast, but very easy to get wrong
  - Try it on all cases, especially trivial ones
  - Look for **invariants**: statements that must be true of any valid list
  - Debug your code by checking invariants
    - In C/C++, this is done via *assert*
    - Most languages have a facility like this built in
      - But if not, you can just write your own!

Memory allocation

- We've assumed the hardware supplied us with a huge array M
  - When we need more storage, we just grab locations at the end
    - Keep track of next free memory location
  - What can go wrong?
    - Consider repeatedly adding, deleting an item
- Notice that when we delete items from a linked list we simply changed pointers so that the items were inaccessible
  - But, they still waste space!

Storage reclamation

- Need to figure out certain locations can be re-used (“garbage”)
  - If too conservative, programs will run slower and slower
    - “memory leak”
  - If it’s too aggressive, your program will crash
    - BSOD or Sad Mac



- Suppose your linked list was corrupted
  - Digression: Why do computers crash when we read/write an arbitrary location? Must they??
  - See: 3131961357
  - 0xbaadf00d

Two basic options

- Computer keeps track of free storage
- Manual storage reclamation
  - Programmer has to explicitly free storage
  - Languages: C, C++, assembler
- Automatic storage reclamation
  - Computer will free storage for you
  - Languages: Matlab, Java, C#, Scheme, SML, Python
- Basic tradeoff is program speed versus programmer effort
  - The computer isn't (usually!) as smart as a programmer

Allocation issues

- Computer keeps a linked list of free storage blocks (“freelist”)
  - For each block, keep size and location
  - When asked for new storage, get from freelist
- Surprisingly important question:
  - Which block do you supply?
- Different sized blocks
  - Actually, several different lists (for efficiency)
    - “Buddy block” system, see: http://www.memorymanagement.org/

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:
  - Everything should be freed when it is no longer going to be used
  - And, it should be freed exactly once!
  - Also want to minimize fragmentation
- Serious C programmer write their own memory allocator!

Automatic storage reclamation

- First challenge: figure out what memory locations are in use by the programmer
  - They are the ones that the programmer can get to (i.e., in a linked list currently in use)
  - Anything that has a name the programmer can get to
  - Start with something that has a name, then chase pointers
  - Some programs can access everything…
    - These languages don't allow such programs!

How to garbage collect?

- Need to distinguish a value from a pointer
  - Garbage collector doesn't know anything about the programmer's data representation
- Usual solution: use highest-order bit
  - The pointer to location 3 is represented as $2^n + 3$, where n is usually 31 (word size – 1)
  - Some computers had hardware support to tell pointers from values
    - Nice idea that died out in the 90's

Simple algorithm: mark-sweep

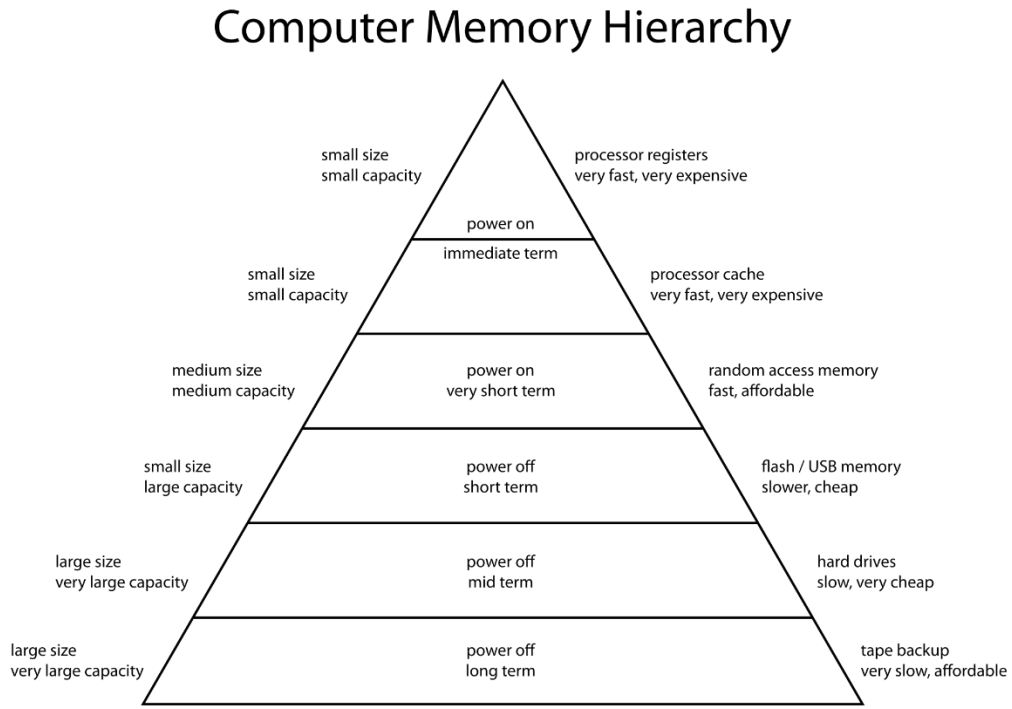
- Chase the pointers from the root set until you are done
  - Modifying everything as you go (“mark”)
    - Typically, set a bit
- Everything not marked is garbage, goes back on the free list
- Problem: typically a lot more garbage than non-garbage, and need to examine all the garbage with mark-sweep
- Python uses reference counting, or more complex schemes

Speed of memory access

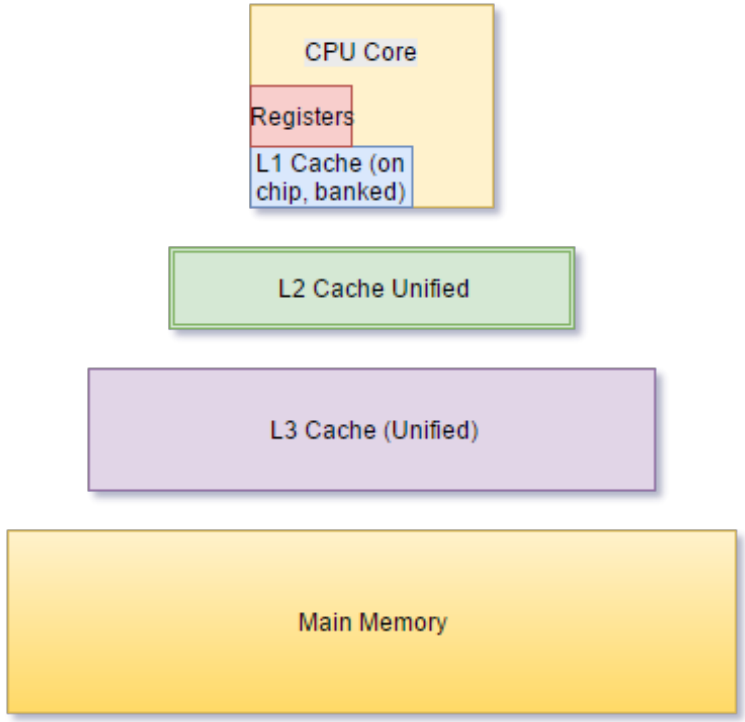
- We made the simplifying assumption that it's constant
  - It's not, and this matters. A lot.
  - Physics dictates memory is small and fast vs large and slow
- There are huge differences in speed depending on how a program accesses memory
  - Certain serious programmers obsess over this (e.g. video games)

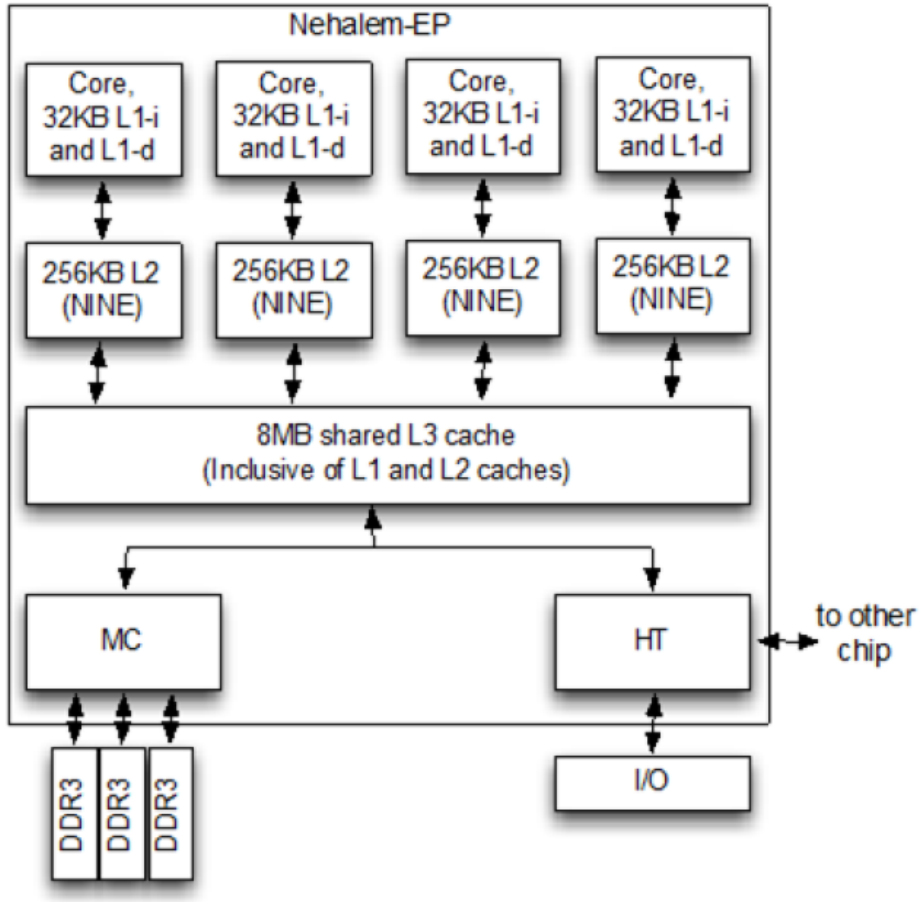
Memory heirarchy

- Basic idea: look for data in small fast memory first
- On a miss, look in slower bigger memory
  - Write is similar to read



Real example: Intel Nehalem





- Example timings (not real):
  - main memory = 50 ns, L1 = 1 ns (10% miss rate), L2 = 5 ns (1% miss rate), L3 = 10 ns (0.2% miss rate) [Wikipedia]

Practical consequences

- Data structures and algorithms that are smart about memory access can be WAY faster than ones that are dumb
- Key idea: exploit locality in space and time
- Consequence of cache layout, and refill strategy
  - Classical example: array access
    - Row major or column major order?
  - Prim versus Kruskal?
- Cache management is quite complex on modern hardware