Intro to Algorithms and Data Structures for Computational Scientists

Edward Valeev

Department of Chemistry
Virginia Tech
Blacksburg, VA

Last updated: **July 10, 2014**

Lecture Outline

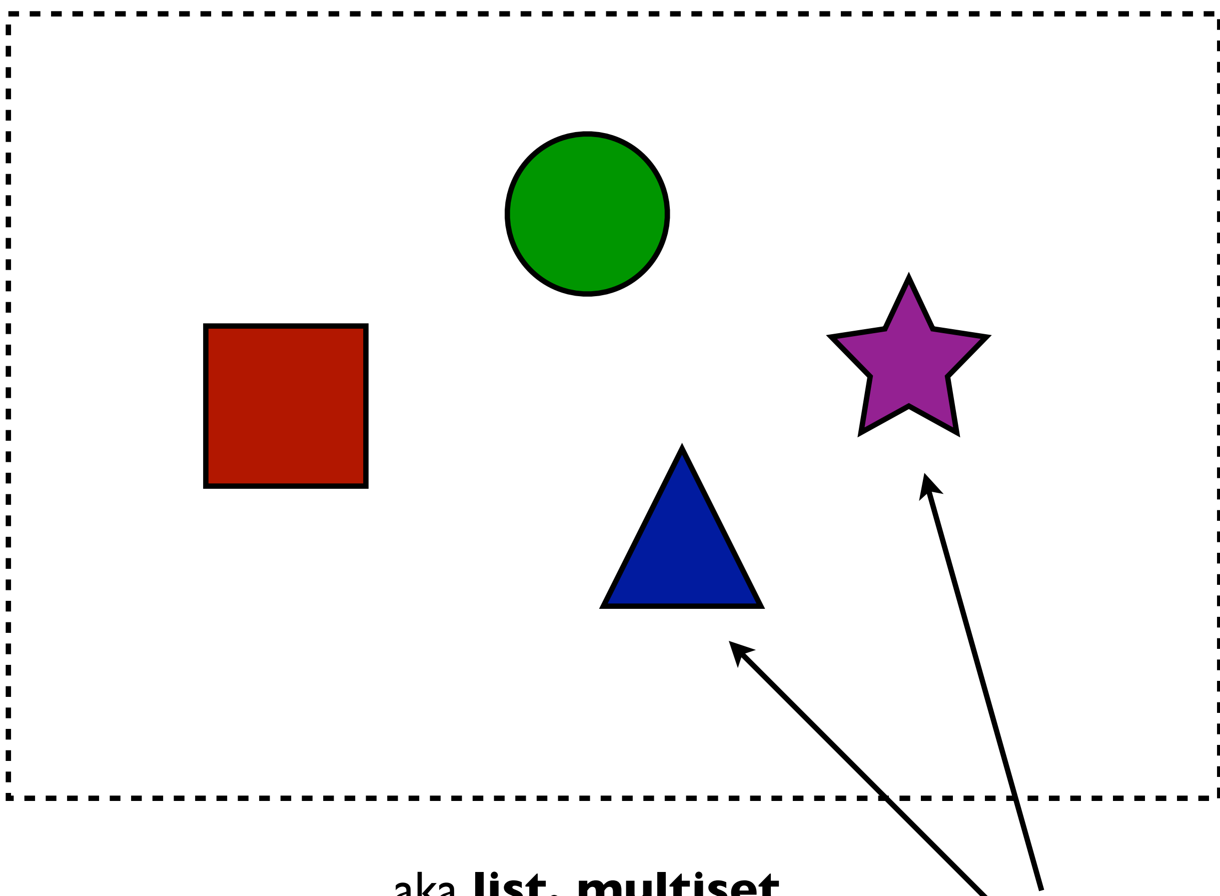
- Definitions
- Data structures
  - basic data structures: lists, sets, arrays, stacks, hash tables, trees, graphs
  - special case: matrices and multidimensional arrays
- Algorithms
  - non-numerical algorithms: sort, search
  - numerical algorithms, linear algebra
- Misc topics
  - numerical properties

Definitions

- Data structure = data + logical relationships between the data
  - Examples: a set of 13 real numbers, a Hamiltonian matrix, a Russian-English dictionary, WWW
- Algorithm = step-by-step precisely-defined recipe for computing output values from input values
  - Examples: Euclid's algorithm (find GCD of 2 numbers), matrix multiplication, "googling"
- There is no best choice! "Best" data structure and "best" algorithm can only be understood relative to the particular model of computer architecture.

Abstract Data Structures

Container



aka **list, multiset**

N.B. same element can appear more than once

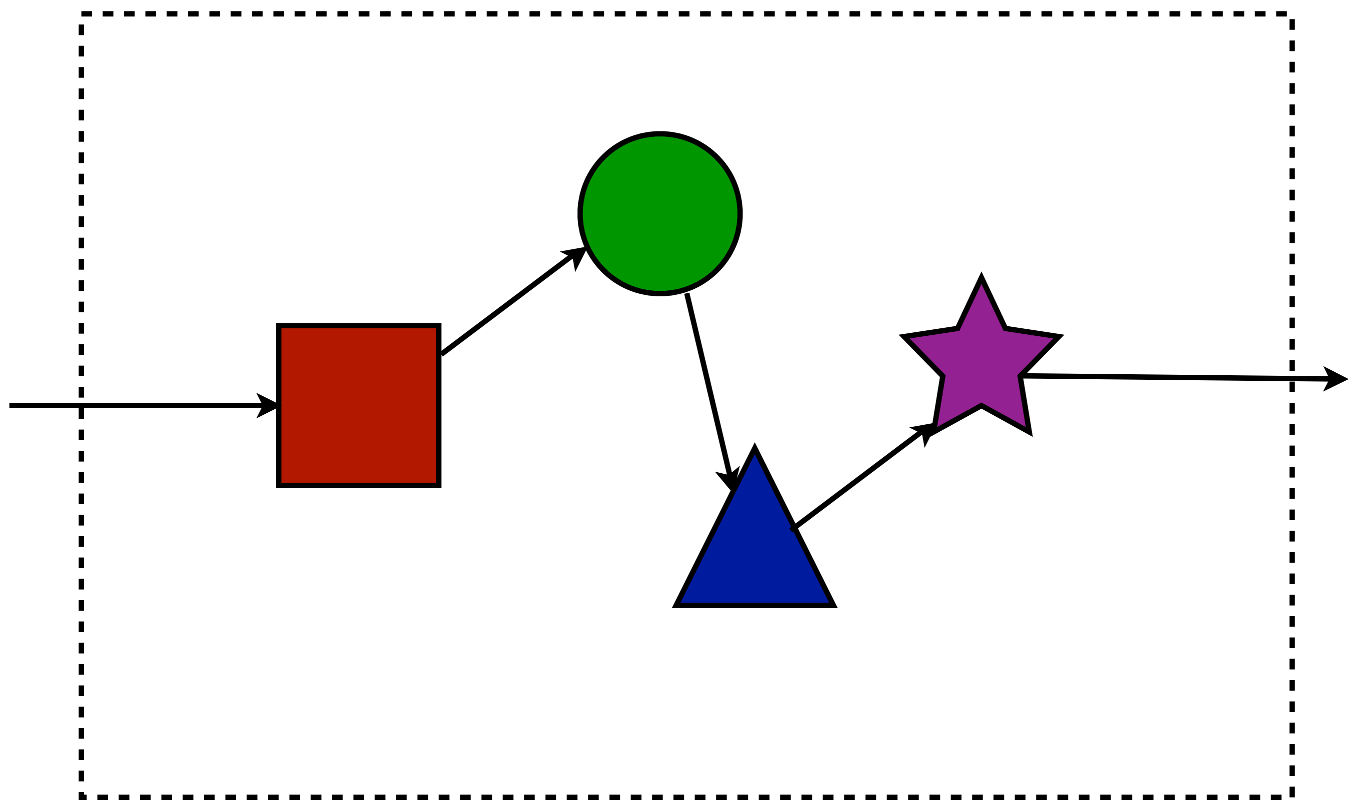
elements/nodes/records/values

Operations

- size()
- iteration over elements (not necessarily unique)
- * remove(X)
- * insert(X)

Abstract Data Structures

Sequence



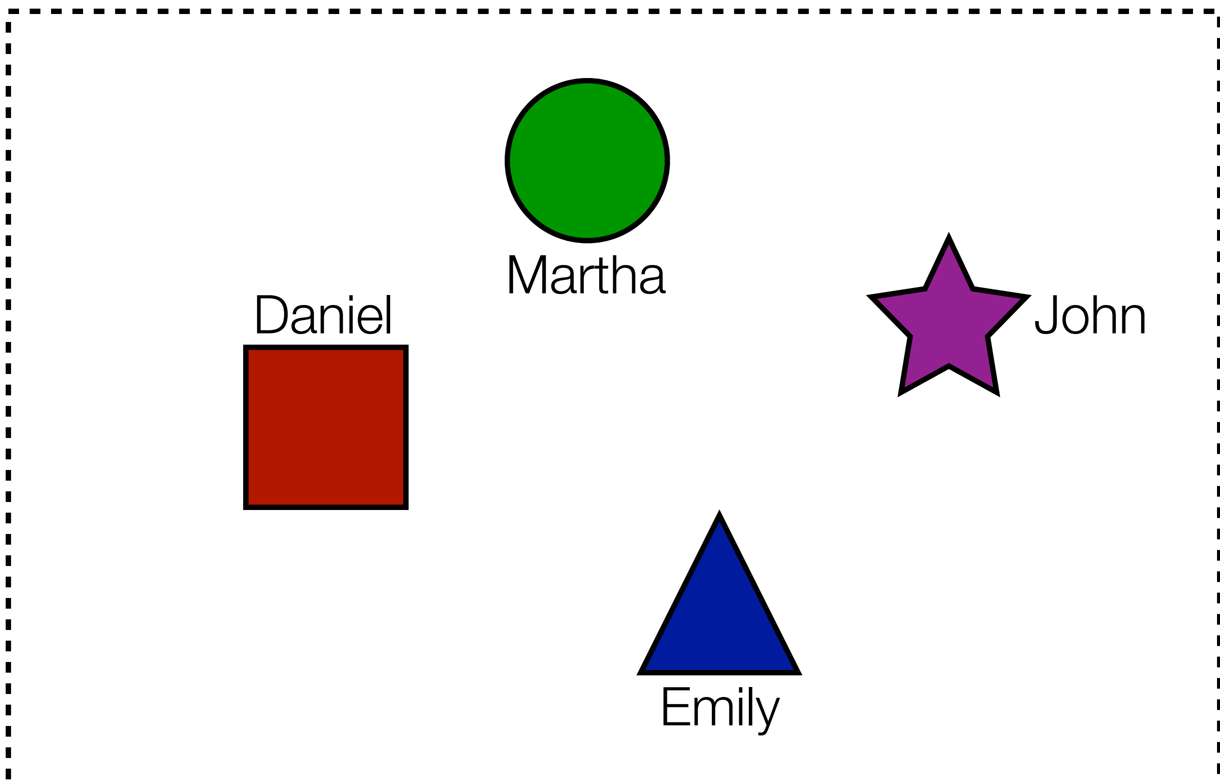
aka **linear list, ordered set**

Operations

- size()
- unique iteration over elements
- remove(p) = removes element pointed to by p
- insert(p, X) = insert X following p
- begin() = return pointer to the first node

Abstract Data Structures

Associative Container



aka **map**

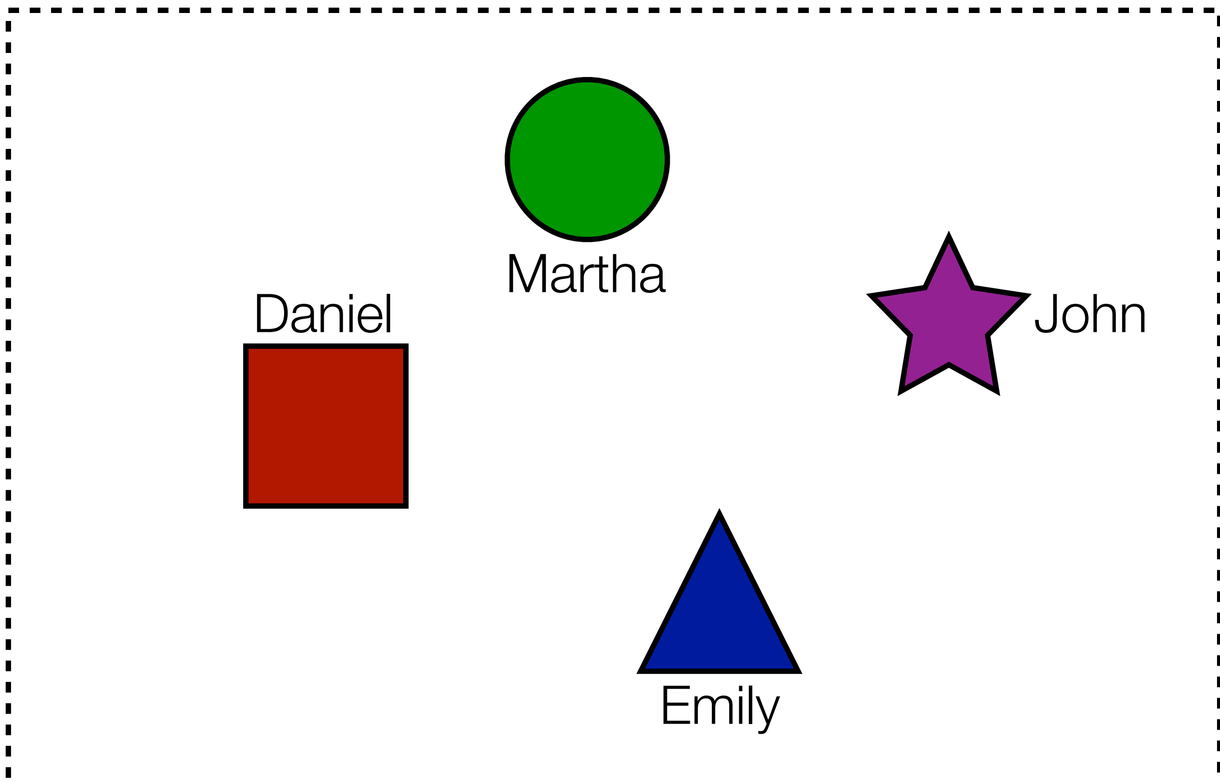
Multiple **Associative Container = same key maps to more than one element**

Additional Operations

- find(K) = return reference to element attached to key K
- count(K) = number of elements whose key is K

Abstract Data Structures

Associative Container



aka **map**

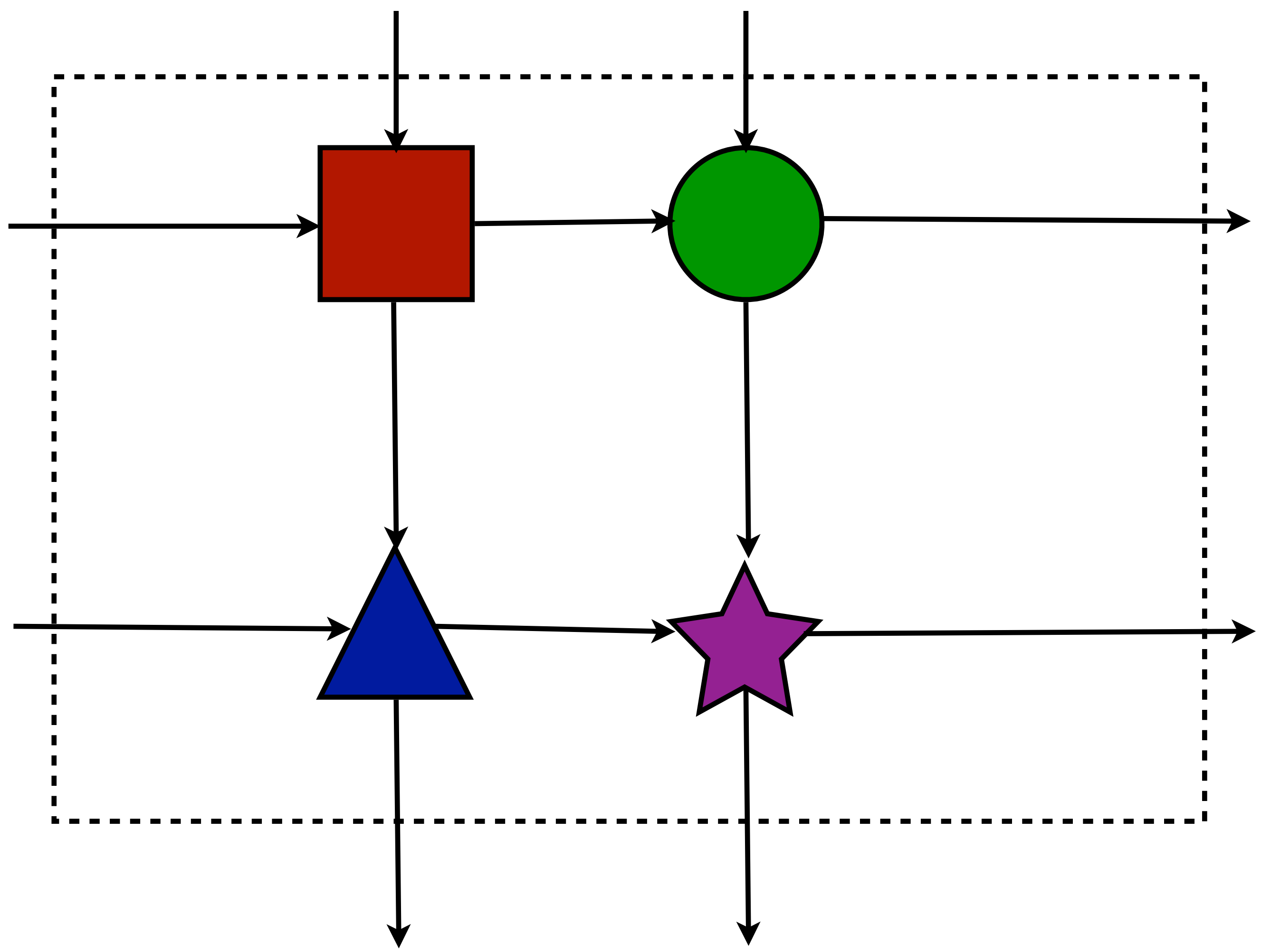
Additional Operations

- find(K) = return reference to element attached to key K
- count(K) = number of elements whose key is K

Can compose other data structures from basic ones

Abstract Data Structures

2-dimensional Array



Operations

- size()
- [i,j] = direct access to element in row i and column j

can be viewed as Simple Associative Container keyed by 2-tuples

Concrete Data Structures

- Concepts you encounter in your work often have natural affinity to particular abstract data structures
  - array of basis function values at a particular point, sequence of trajectory snapshots = Sequence
  - database of computations, list of atoms contributing to a particular orbital = Associative Container
- But, same set of data can be viewed as several abstract data structures.
- An abstract data structure can be implemented in several concrete ways.

Concrete Data Structures

- Sequence
  - vector
  - (linked) list
  - stack
- Associative Container
  - hash table
  - (multi)map

Concrete Data Structures

Vector

aka **(1-d) array**

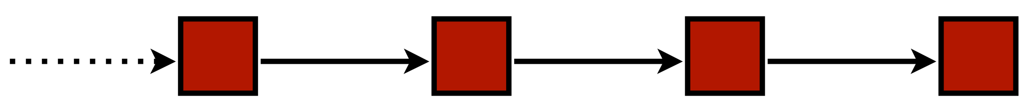
C++ example

```
#include <vector>

std::vector<double> v3(10); // vector of 10 uninitialized elements
v3.resize(20);
v3[5] = 7.0;
```

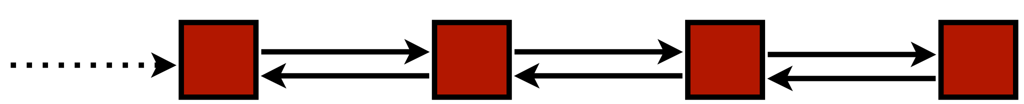
- sequential storage of elements in memory = v[i] is next to v[i-1] and v[i+1]
- pointer to element i = pointer to element 0 + i * sizeof(element)
- hence cheap (“O(1) cost”) access to each element
- expensive (“O(n) cost”) element insertion/deletion
- no storage overhead

Concrete Data Structures

Singly-Linked List



Doubly-Linked List



C++ example

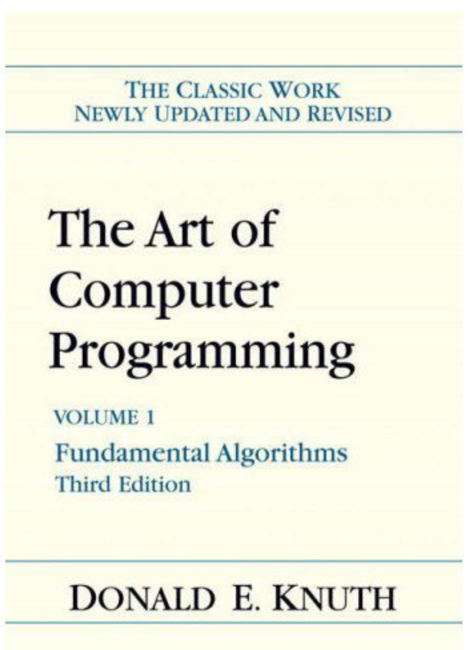
```
#include <list>

std::list<double> l1; // empty C++ (doubly-linked) list
                      // don't forget to #include <list>
l1.push_back(1.0);             // l1 = { 1.0 }
l1.push_front(2.0);            // l1 = { 2.0, 1.0 }
l1.insert(++l1.begin(), 3.0); // l1 = { 2.0, 3.0, 1.0 }
l1.erase(--l1.end());          // l1 = { 2.0, 3.0 }
l1.clear();                    // l1 = {}
```

- non-sequential storage of elements in memory
- each node stores the value + the pointer(s) to the neighbor
- O(1) access to next element
- O(n) access to the i-th element
- O(1) insertion/deletion
- O(n) storage overhead

Concrete Data Structures

More "Sequences": Stack, Queue, Deque

Figure from Knuth *The Art of Computer Programming: Vol 1* (3rd ed), Addison Wesley (1997)



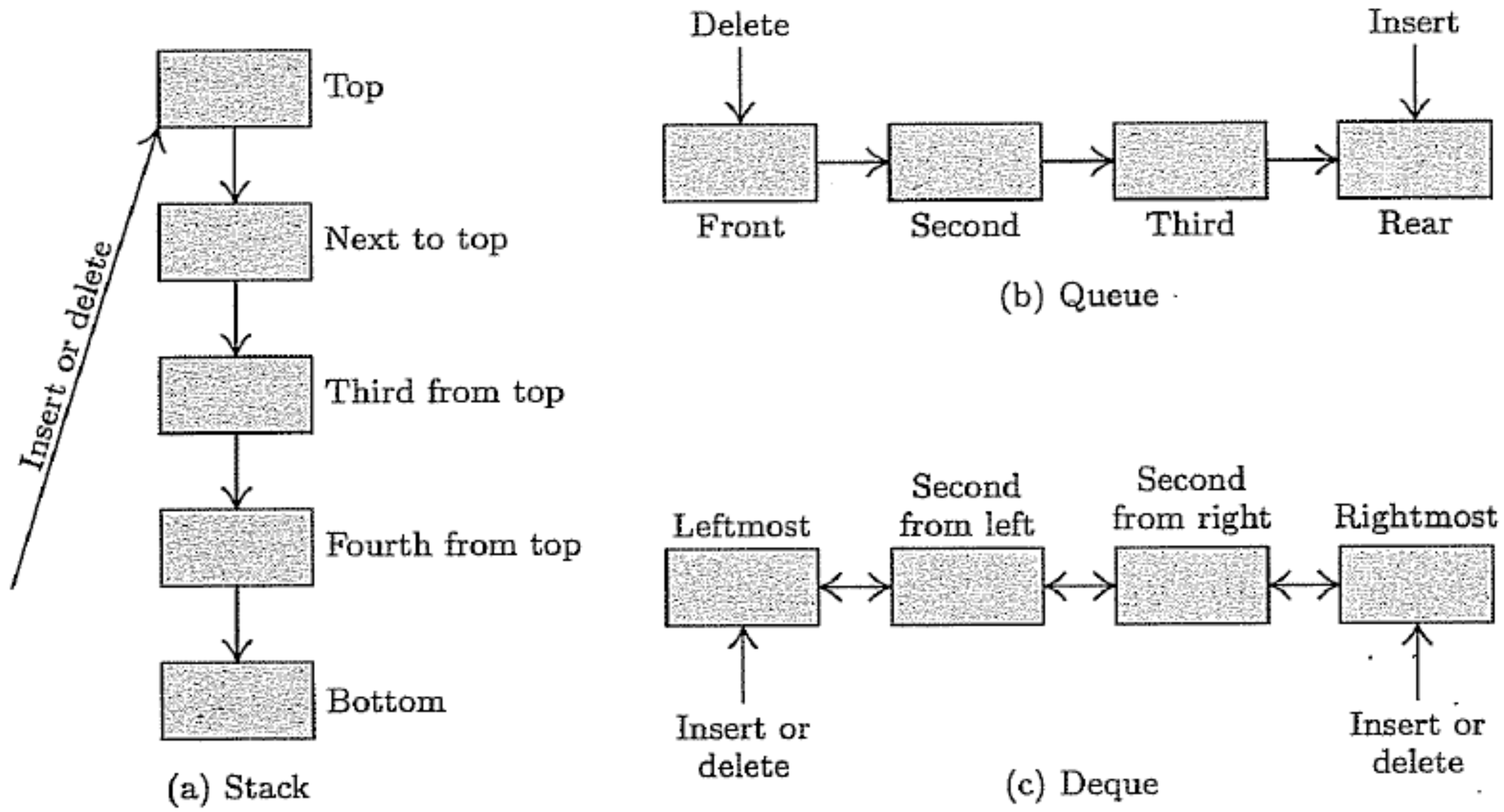


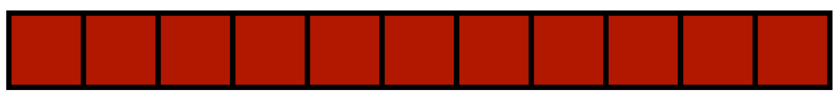
Fig. 3. Three important classes of linear lists.

Part of standard C++: std::stack, std::queue, and std::deque

Concrete Data Structures

Hash Table

pos({key,value}) = hashfunc(key)



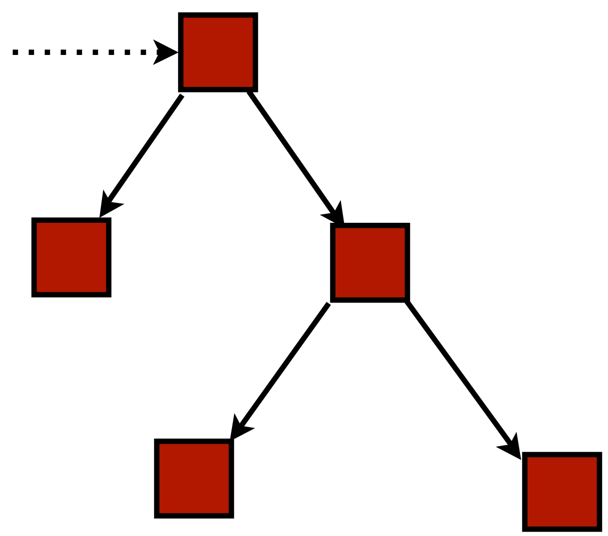
Hashed containers are part of 2011 C++ standard (see std::unordered_map and std::unordered set)

```
#include <iostream>
#include <unordered_map>

std::unordered_map<std::string, int> h1; // maps string -> int
                                          // std::hash<std::string> is
                                          // a standard hash function
h1["january"] = 31;           // "january" => 31
h1["february"] = 28;          // "february" => 28
h1["march"] = 31;             // "march" => 31
h1.clear();                   // h1 is empty
```

- no unique iteration order
- each node stores the key and value
- On average: O(1) access to the i-th element, insertion, deletion
- At worst: O(n) access to the i-th element, insertion, deletion
- no storage overhead
- **good hash function is key!**

Concrete Data Structures

Tree

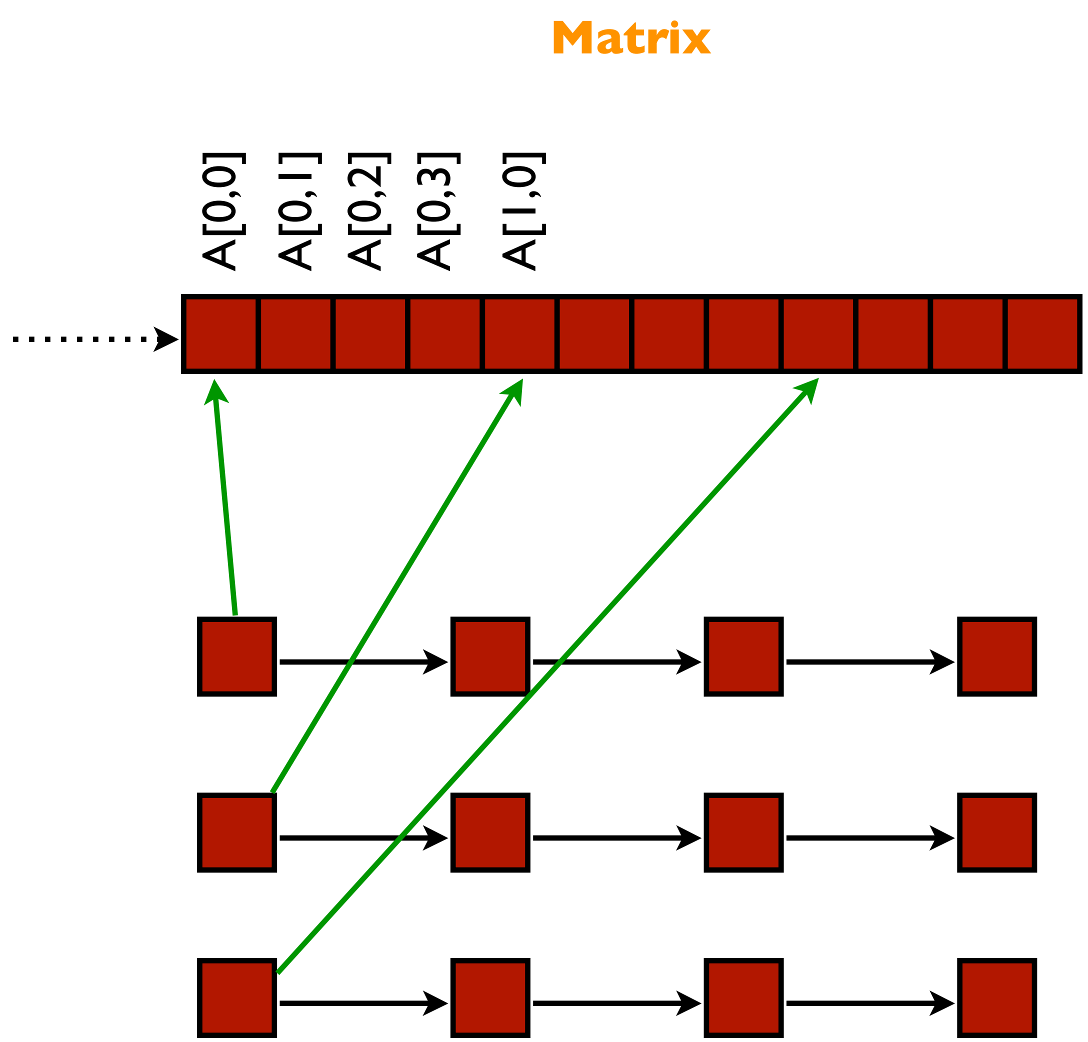


No standard C/C++ implementation (but many containers are implemented in terms of trees)

see Boost for implementation of a Graph

- no unique iteration order
- each node stores the value + the pointer(s) to the children
- O(1) access to next element
- O(log n) access to the i-th element in a balanced tree
- O(1) insertion/deletion
- O(n) storage overhead

Multidimensional Arrays



$$pos(A[i,j]) = pos(A[0,0]) + i * n_{col} + j$$

standard C/C++ implementation is *array of pointers to rows*
see also Eigen, Elemental, and other proper C++ libraries

- sequential storage of elements in rows, not columns
- O(1) access to next element
- O(1) access to any element (but some arithmetic involved)
- no storage overhead
- can be generalized to any number of dimensions, as well as symmetries

Interlude: Big O Notation

- Asymptotic behavior of algorithms (e.g. when problem size become large) is useful to characterize in rough terms using the Big O (and related) notation

- $O(g(x))$ is a set of functions for whom $g(x)$ is an asymptotic upper bound

  - working definition: $f(x) = O(g(x))$ ("f(x) is big oh of g of x") if there exist positive x_0 and c such that $0 \leq f(x) \leq c\, g(x)$ for all $x \geq x_0$

  - $7x^2 - 20x + 1 = O(x^2)$

  - $1000 = O(1)$

  - does not imply *tight* upper bound: $x^2 = O(x^3)$

Interlude: Big O Notation

- $\Omega(g(x))$ is a set of functions for which $g(x)$ is an asymptotic lower bound
  - formal definition: $f(x) = \Omega(g(x))$ ("f(x) is big omega of g of x") if there exist positive x_0 and c such that $c\, g(x) \leq f(x)$ for all $x \geq x_0$
  - $7\, x^2 - 20x + 1 = \Omega(x^2)$
  - $1000 = \Omega(1)$
  - $x^2 = \Omega(x)$

Interlude: Big O Notation

- $\Theta(g(x))$ is a set of functions for whom $g(x)$ is an asymptotic tight bound
  - formal definition: $f(x) = \Theta(g(x))$ ("f(x) is big theta of g of x") if there exist positive x_0 and c_1 and c_2 such that $c_1\, g(x) \leq f(x) \leq c_2\, g(x)$ for all $x \geq x_0$
  - $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$ implies $f(x) = \Theta(g(x))$
  - $7\, x^2 - 20x + 1 = \Theta(x^2)$
  - $1000 = \Theta(1)$
  - Informal recipe: only keep the leading order term, ignore its prefactor
  - Usually when people say $O()$ they mean $\Theta()$!!!

Classic Algorithms

Sort

Input: $\{i_1, i_2, i_3 \ldots. i_n\}$

Output: $\{i'_1, i'_2, i'_3 \ldots. i'_n\}$ = a permutation of $\{i_1, i_2, i_3 \ldots. i_n\}$
such that $i'_1 \leq i'_2 \leq i'_3 \leq \ldots. \leq i'_n$

Many Algorithms!
let's consider a few to understand how to analyze algorithms

Classic Algorithms

Bubble Sort

6 5 3 1 8 7 2 4

(C) 2011 Wikimedia Commons
http://en.wikipedia.org/wiki/File:Bubble-sort-example-300px.gif

```
swapped = true
iter = 0
while (swapped)
  swapped = false
  for i=1 .. A.size()-1-iter
    if A[i] < A[i-1]
      swap A[i] and A[i-1]
      swapped = true
  iter = iter + 1
```

Stable, in-place, simple
Efficient for vectors and lists
at best O(n)
on average $O(n^2)$
at worst $O(n^2)$

Classic Algorithms

Insertion Sort

6 5 3 1 8 7 2 4

for i=1 .. A.size()-1
  insert A[i] into sorted sequence A[0]..A[i-1]

Stable, in-place, simple
Efficient for lists
at best O(n)
on average $O(n^2)$
at worst $O(n^2)$

Classic Algorithms

Merge Sort

6 5 3 1 8 7 2 4

(C) 2011 Wikimedia Commons
http://en.wikipedia.org/wiki/File:Merge-sort-example-300px.gif

Example of a *divide-and-conquer* algorithm

1. *recursively* subdivide sequence into unit size subsequences
2. merge resulting sorted subsequences

Stable
Efficient for vectors and lists
$O(n \log n)$
Can be in-place at $O(n (\log n)^2)$

Classic Algorithms

Combinatorial Search

Input: $A=\{i_1, i_2, i_3 \ldots\ i_n\}$, and boolean function $f(x)$

Output: (pointer to) node k or set of nodes $\{k_1, .. k_m\}$ for which $f(i_k)$ = true

Examples

- **Element search:** search node whose value matches *search key* a, i.e. $f(x) = a$
- **Min (max) search:** search node whose value is min(A) (or max(A))
- **Subset search:** search nodes whose values match the search key $\{a_1 .. a_m\}$
- etc.

Combinatorial Search is a subset of general Search problem, that includes solving equations, finding minima of functions, etc.

Numerical Algorithms: Linear Algebra

Vector Operations

DOT

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i^* b_i$$

AXPY

$$\mathbf{y} = \mathbf{y} + a\mathbf{x}$$

$$y_i = y_i + a\,x_i$$

and many others

Numerical Algorithms: Linear Algebra

Vector Operations

DOT

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i^* b_i$$

AXPY

$$\mathbf{y} = \mathbf{y} + a\mathbf{x}$$
$$y_i = y_i + a\, x_i$$

Performance Considerations

- **Stride-1 access:** good memory locality for vectors (worse for lists!)
- **Independent loop iterations:** natural data parallelism, good vectorization
- **Bandwidth limited:** O(n) MOPs, O(n) FLOPs
    - DDOT: 2 MOPs per 2 FLOPs
    - AXPY: 3 MOPs per 2 FLOPs

performance will largely depend on where the data is located (L2 cache - good; RAM - bad)

Numerical Algorithms: Linear Algebra

Performance of DAXPY vs. vector size (GFLOP/s)

	n	Intel Xeon E5645 “Nehalem” 2.4 GHz (1.3 GHz DRAM) SSE2 peak=9.6	Intel Core I7-3820QM “Ivy Bridge” 2.7 GHz (1.6 GHz DRAM) AVX peak=21.6
data in L1 cache	1024	4.7	11.7
	2048	4.6	11.6
data in L2 cache	4096	2.7	5.4
	8192	2.7	5.3
data in L3 cache	100000	1.8	3.8
data in main memory	10000000	1	1.6

Numerical Algorithms: Linear Algebra

Matrix Multiplication

$$\mathbf{C} = \mathbf{A}\mathbf{B}$$

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

although in practice ... GEMM

$$\mathbf{C} = \alpha \mathbf{A}\mathbf{B} + \beta \mathbf{C}$$

see tomorrow's lecture on BLAS

Numerical Algorithms: Linear Algebra

Matrix Multiplication

```
for i=0 .. n-1
 for j=0 .. n-1
  v = 0.0
  for k=0 .. n-1
   v += A[i,k] * B[k,j]
  C[i,j] = v
```

Performance Considerations

- **Stride-1 access for A but stride-n for B:** bad memory locality
- **Independent loop iterations:** natural data parallelism, good vectorization
- **Bandwidth limited as written:** $O(n^3)$ MOPs, $O(n^3)$ FLOPs

how to improve? Increase *data reuse*

Numerical Algorithms: Linear Algebra

Blocked Matrix Multiplication

```
for I=0 .. n/b-1
 for J=0 .. n/b-1
  v = 0.0
  for K=0 .. n/b-1
   load A[I*b .. (I+1)*b-1, K*b .. (K+1)*b-1] into cache
   load B[K*b .. (K+1)*b-1, J*b .. (J+1)*b-1] into cache
   compute C[I*b .. (I+1)*b-1, J*b .. (J+1)*b-1]
  C[i,j] = v
```

Performance Considerations

- **Stride-1 access for A and B:** good memory locality
- **Independent loop iterations:** natural data parallelism, good vectorization
- **Compute limited as written:** $O(n^3)/b$ MOPs vs. $O(n^3)$ MOPs before

Numerical Algorithms: Linear Algebra

Standard Algorithm: 8 muls + 4 adds

$$\mathbf{C} = \mathbf{A}\mathbf{B}$$

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$$

$$\begin{aligned}
\mathbf{C}_{11} &= \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} \\
\mathbf{C}_{12} &= \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\
\mathbf{C}_{21} &= \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} \\
\mathbf{C}_{22} &= \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22}
\end{aligned}$$

Numerical Algorithms: Linear Algebra

Strassen Algorithm: 7 muls + 18 adds

$$\mathbf{C} = \mathbf{A}\mathbf{B}$$

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22}) \\
\mathbf{M}_4 &= \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22} \\
\mathbf{M}_6 &= (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12}) \\
\mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})
\end{aligned}$$

$$\begin{aligned}
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

Numerical Algorithms: Linear Algebra

Strassen Algorithm: 7 muls + 18 adds

$$\mathbf{C} = \mathbf{AB}$$

$$f(n) \equiv \text{cost of multiplication of rank n matrices}$$

$$f(n) \approx 7\, f(n/2)$$

solving the recursive equation yields

$$f(n) = \mathcal{O}(n^{\log_2 7}) \approx \mathcal{O}(n^{2.8})$$

faster than standard algorithm for large matrices!
but slower for small matrices (e.g. for n=2 standard costs 12 FLOPs but Strassen costs 25 FLOPs)

Numerical Stability of Algorithms

Example: linear system

what we think we are solving:

$$\mathbf{Ax} = \mathbf{b}$$

what we are actually solving (N.B. discarding noise in A):

$$\mathbf{A}(\mathbf{x} + \mathbf{e}') = \mathbf{b} + \mathbf{e}$$

formally:

$$\begin{aligned} \mathbf{x} &= \mathbf{A}^{-1}\mathbf{b} \\ \mathbf{e}' &= \mathbf{A}^{-1}\mathbf{e} \end{aligned}$$

measures how relative error in B relates in relative error in X

$$\text{condition number} = \frac{||\mathbf{e}'||/||\mathbf{x}||}{||\mathbf{e}||/||\mathbf{b}||} = ||\mathbf{A}^{-1}||\,||\mathbf{A}||$$

$$\text{condition number}(\mathbf{A}) = \frac{\sigma_{\max}}{\sigma_{\min}}$$

singular values

similar analysis works for any other input -> output scheme

Questions