DS221: Introduction to Scalable Systems

Topic: Algorithms and Data Structures

About this Course

- Designed as an introductory course on algorithms and systems
  - Computer Architecture
  - Data Structures, Algorithms (DSA)
  - Big Data Systems: Concepts and Programming
  - Parallel System: Concepts and Programming
- Covers **breadth**, not *depth*. Precursor to:
  - Scalable systems for data science, DS256
  - Parallel programming, DS295
  - Algorithmic foundations of big-data biology, DS202

Prerequisites

- Accessible to non-computer science UG Majors
- Requires some prior *programming knowledge*
  - We will NOT teach programming
- We will use C++ for DSA
  - Makefile, Standard Template Library
  - IDE (Visual Studio Code), GitHub
  - CoPilot for Visual Code, Prompt engineering

Grading Scheme (DSA, Big Data Systems)

- Assignments [30 points]
  - 1 DSA Programming Assignments (20 points)
  - 1 Quiz (10 points)
- Final Exam [20 points]

Schedule

- Data Structures and Algorithms
  - Lectures: Aug 21 – October 7
  - Programming Assignment 1 *(to be posted next week)*
  - *Quiz (Date TBD)*
- Big Data Concepts
- Tutorial sessions on Wednesdays/Thursdays/Fridays 5-6pm (Will setup a poll today)
  - Online (will be recorded) + In-person office hour (bring your concerns)

Topics

- Introduction to data structures, lists, arrays, sparse matrix
  - Linux Shell/C++/Makefile/VS Code, sign up for free copilot
- Algorithmic analysis and complexity
  - C++ STL
- Prompt Engineering and using Copilot to design code
- Stacks, Queues
  - A1 Q&A
- Priority queues, Tree
- Hashmap, binary tree, b-tree
  - A1 Q&A
- Graphs
  - PySpark Hands On
- Algorithmic design, Concurrency
- Introduction to Big Data Platforms

IISc POLICY FOR ACADEMIC INTEGRITY

- **Plagiarism**
  - Reproducing, in whole or part, text, figures or data from a reports, books, internet, someone else, etc.
  - Taking material from internet sites, class notes, etc. and using it in one's class reports, assignments, etc. without citing the original source.
- **Cheating**
  - Copying during exams, homework assignments, reports.
  - Allowing or facilitating copying by someone else.
  - Using unauthorized material, copying, collaborating when not authorized, and purchasing material.
  - Fabricating (making up) or falsifying (manipulating) data

https://iisc.ac.in/about/student-corner/academic-integrity/

IISc POLICY FOR ACADEMIC INTEGRITY

- **Penalties**
  - Zero points in assessment
  - Grade point drop
  - F grade
  - Expulsion

https://iisc.ac.in/about/student-corner/academic-integrity/

Default Instructions

- You must complete all your assessments independently, without any external assistance from online/external sources, prior notes, etc., or collaboration with other students/people.
- You must not share, look at solutions from others or, copy from external/generative AI sources
- If you took help from anyone/any external source, you should clearly make cite it at the end of your report under "External Attribution"
- Any exceptions to this rule (e.g., using Copilot) will be explicitly mentioned in any assessment

Questions?

http://cds.iisc.ac.in/courses/ds221

Data Structure, Algorithms and Data Systems

Class Resources

- Assessments mostly based on class lectures
- Other resources you can refer to:
  - Data Structures, Algorithms, and Applications in C++, Sartaj Sahni
    - http://www.cise.ufl.edu/~sahni/dsaac/
  - The C++ Programming Language, 3rd Edition, Bjarne Stroustrup
  - C++ Standard Template Library, http://www.cplusplus.com/reference/stl/
  - THE ART OF COMPUTER PROGRAMMING (Volume 1 / Fundamental Algorithms), Donald Knuth
  - Introduction to Algorithms, Cormen, Leiserson, Rivest and Stein
  - www.geeksforgeeks.org/data-structures/

L1: Introduction

Concepts

- **Algorithm**: Outline, the essence of a computational procedure, with step-by-step instructions
- **Program**: An implementation of an algorithm in some programming language
- **Data structure**: Organization of data needed to solve the problem (array, list, hashmap)
- **Algorithmic Analysis**: The expected behavior of the algorithm you have designed, before you run it
- **Empirical Analysis**: The behavior of the program that implements the algorithm, by running it

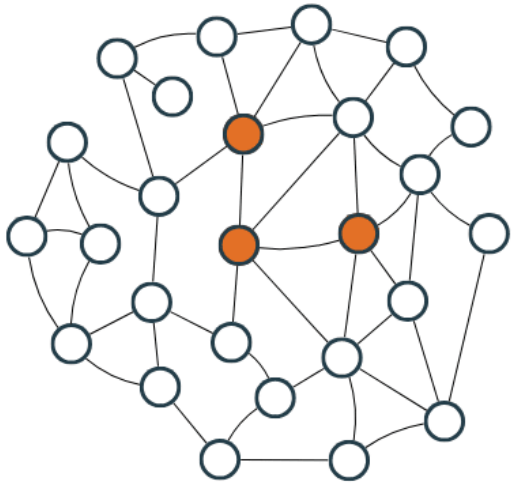
Why not just run it and see how it behaves?

Limitation of Empirical Analysis

- Need to implement the algorithm
  - Time consuming
- Cannot exhaust all possible inputs
  - Experiments can be done only on a limited to set of inputs
- May not be indicative of running time for other inputs
  - Harder to compare two algorithms
- Same hardware/environments needs to be used
- *But useful to perform and compare output with ground truth*

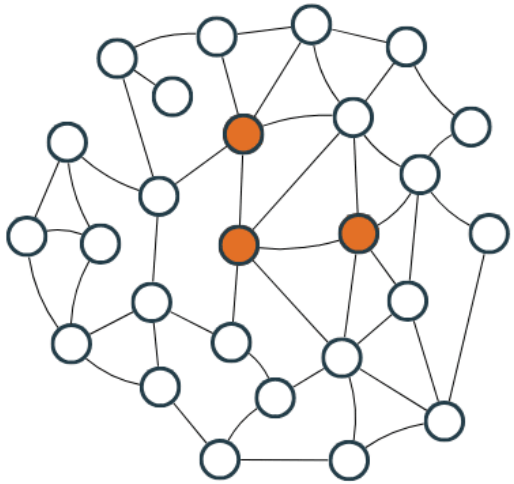
Example: Your test case for a graph algorithm

Suppose the programming assignment states that your code must scale to graphs with ~30 vertices



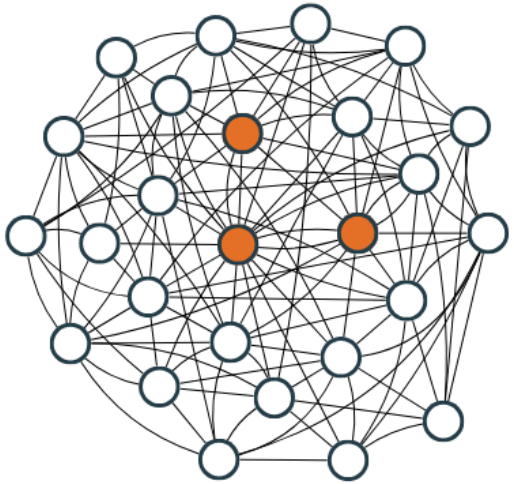
Example: Your test case for a graph algorithm

Suppose the programming assignment states that your code must scale to graphs with ~30 vertices



Test case (during evaluation)

It can happen that the same code does not finish within the time limit



How do we design an algorithm?

- Intuition
- Mixture of techniques, design patterns
- Experience (body of knowledge)
- Data structures, analysis

How do we implement a program?

- Preferred High Level Language, e.g. C++, Java, Python
- Map algorithm to language, retaining properties
- Use native data structures, libraries

How do we design an algorithm?

- Intuition
- Mixture of techniques, design patterns
- Experience (body of knowledge)
- Data structures, analysis

How do we implement a program?

- Preferred High Level Language, e.g. C++, Java, Python
- Map algorithm to language, retaining properties
- Use native data structures, libraries

Then why learn about basic data structures?

Algorithm, Data Structure & Language are interconnected

- Algorithms based on specific data structures, their behaviour
- Algorithms are limited to the features of the programming language
  - Procedural, Functional, Object oriented, distributed
- Data structures may/may not be natively implemented in language
  - Java Collections, C++ STL, NumPy

Basic Data Structures

Lists

Collections of Data

- Data Structures to store collections of data items of same type
  - Items also called elements, instances, values…depending on context
- Primitive types can be boolean, byte, integer, etc.
- Complex types can be user or system defined objects, e.g., node, contact, vertex
- Operations on the collection
  - Standard operations to create, modify, access elements
- Properties of the collection
  - **Invariants that must be maintained**, irrespective of operations performed
- **Challenge**: Understand how to pick the right data structure for your application!

Collections of Data

- Can have different implementations for same abstract data type
  - All offer **same** operations and invariant guarantees
  - Differ in performance (space/time complexity)
- **Challenge**: Understand how to pick the right implementation!
- Also, do we need to have different copies of code for different data types (e.g., integer, floats)?

Try yourself!

- Learn **templates/generics**. In many collections, the item type does not matter for invariants and operations, and can be replaced by a place-holder type "T".
- Learn C++ **Standard template library (STL)**. Read up examples of abstract collections and their implementations.
- http://www.cplusplus.com/reference/stl/

Linear List *(abstract data type)*

- Properties
  - **Ordered** list of items…precedes, succeeds; first, last
  - **Index** for each item…lookup or address item by index value
  - **Well-defined size** for the list at a point in time…can be empty, size may vary with operations performed
  - Items of **same type** present in the list
- Operations
  - Create, destroy
  - Add, remove item
  - Lookup by index, item value
  - Find size, check if empty
  - *Precise name of operation may vary with language, but semantics remain same/similar….READ THE DOCS!*

Type = int, Size = 7

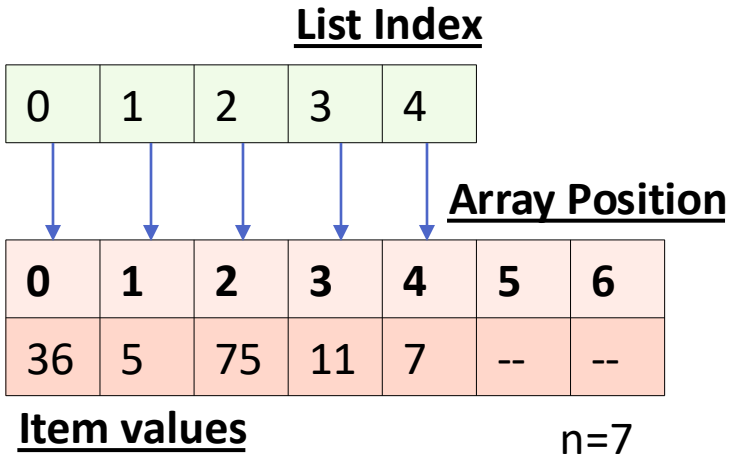
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
Item	36	5	75	11	7	19	-1

1-D Array *(implementation of list)*

- List implementation using arrays, in a prog. language
- Arrays are contiguous memory locations with fixed capacity
  - Contiguous locations mean **locality** matters!
  - **Capacity** is different from **size**. Size is current number of items in list. Capacity denotes max possible size.
- Allow elements of same type to be present at specific positions in the array
  - Position is the **offset** from the start of array memory location, while accounting for **data type size**

Mapping Function

- Index in a **List** can be mapped to a Position in the **Array**
  - **Mapping function** from index to position
- Say **n** is the capacity of the array
- Simple mapping
  - **position**(index) = index



List Operations

- item **get**(index)
- void **set**(index, item)
- void **insert**(index, item)
- void **erase**(index)

- int **size**()
- int **capacity**()
- boolean **isEmpty**()
- int **find**(item)

List Operations using Arrays

- void **create**(initCapacity)
  - Create array with initial capacity *(optional hint)*
- void **set**(index, item)
  - Use mapping function to set value at position
  - Sanity checks?
- item **get**(index)
  - Use mapping function to set value at position
  - Sanity checks?

```
class Array implements List {
// Array impl of list with index starting at 1

      int arr[]                 // backing array for list
      int capacity              // current capacity of array
      int size                  // max. occupied position in list

      /**
      * Create an empty list with optional
      * initial capacity provided. Default capacity of 15
      * is used otherwise.
      */
      void create(int _capacity){
        capacity = _capacity > 0 ? _capacity : 15
        arr = new int[capacity]     // create backing array
        size = 0                    // initialize size
      }
```

```
// assuming pos = index-1 mapping fn.
void set(int index, int item){
     if(index > capacity) { // grow array, double it at least

               arrNew = int[MAX(index, 2*capacity)]
               // copy all items from old array to new
               // source, target, src start, trgt start, length
               copyAll(arr, arrNew, 0, 0, capacity)
               capacity = MAX(index, 2*capacity) // update var.
               delete(arr)     // free up memory

               arr = arrNew    // update var.
     }
     if(index < 1) {
               cout << "Invalid index:" << index << "Expect >=1"
     } else {
               int pos = index - 1
               arr[pos] = item
               size++
     } // end if
   } // end set()
} // end List
```

Try yourself!

Implement **get**(index)

List Operations using Arrays

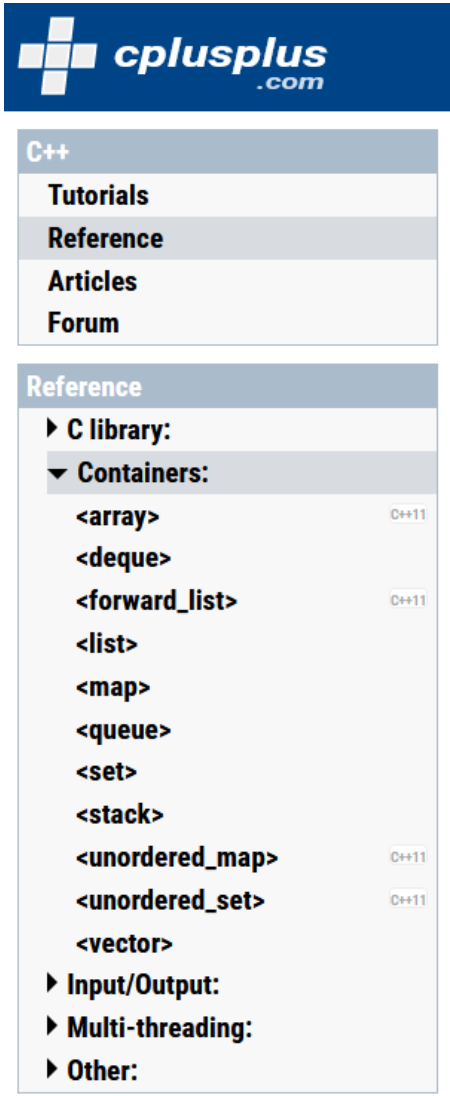
- int **find**(item)
  - Get “first” index of item with given value
  - Sanity checks?
- void **erase**(index)
  - May replace the item with a NULL *or* shift all items at (index+1, index+2,…) left by 1
- void **insert**(index, item)
  - Insert item at index and shift items to right by one position

List Operations using Arrays

- Increasing capacity
- Start with initial capacity given by user, or default
- When capacity is reached
  - Create array with more capacity, e.g. double it
  - Copy values from old to new array
  - Delete old array space
- Can also be useful to shrink space
  - Why?
- **Pros & Cons of List using Arrays**

Do I have to implement all of these?

- *No!*
- Most languages have efficient implementation for these
  - But you should read their documentation
  - *Maybe even look at their implementation!*
- Java Collections
- C++ Standard Template Library
  - "Containers" to implement common data structures



https://cplusplus.com/reference/stl/

public member function

std::array::at

<array>

```
reference at ( size_type n );const_reference at ( size_type n ) const;
```

Access element

Returns a reference to the element at position ***n*** in the array.

The function automatically checks whether ***n*** is within the bounds of valid elements in the container, throwing an out_of_range exception if it is not (i.e., if ***n*** is greater than, or equal to, its size). This is in contrast with member operator[], that does not check against bounds.

Parameters

n

Position of an element in the array.
If this is greater than, or equal to, the array size, an exception of type out_of_range is thrown.
Notice that the first element has a position of 0 (not 1).
Member type size_type is an alias of the unsigned integral type size_t.

Return value

The element at the specified position in the array.

If the array object is const-qualified, the function returns a const_reference. Otherwise, it returns a reference.

Member types reference and const_reference are the reference types to the elements of the array (see array member types).

Example

```
// array::at
#include <iostream>
#include <array>

int main ()
{
  std::array<int,10> myarray;

  // assign some values:
  for (int i=0; i<10; i++) myarray.at(i) = i+1;

  // print content:
  std::cout << "myarray contains:";
  for (int i=0; i<10; i++)
    std::cout << ' ' << myarray.at(i);
  std::cout << '\n';

  return 0;
}
```

Output:

```
myarray contains: 1 2 3 4 5 6 7 8 9 10
```

Complexity

Constant.

Iterator validity

No changes.

Data races

The reference returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

Fixed Size **Array** "Container" in STL

- Container version of *native* array[ ]
- Allows compatibility with other container data structures

Read the Docs!

Read the Code!

```
226       constexpr const_reference
227       at(size_type __n) const
228       {
229         // Result of conditional expression must be an lvalue so use
230         // boolean ? lvalue : (throw-expr, lvalue)
231         return __n < _Nm ? _M_elems[__n]
232           : (std::__throw_out_of_range_fmt(__N("array::at: __n (which is %zu) "
233                                                ">= _Nm (which is %zu)"),
234                                            __n, _Nm),
235              _M_elems[__n]);
236       }
```

https://gcc.gnu.org/onlinedocs/gcc-13.2.0/libstdc++/api/a00047_source.html

https://cplusplus.com/reference/array/array/at/

std::array::fill

```
void fill (const value_type& val);
```

Fill array with value

Sets ***val*** as the value for all the elements in the array object.

Parameters

val

Value to fill the array with.
Member type value_type is the type of the elements in the container, defined in array as an alias of its first template parameter (T).

Return value

none

Example

```
1  // array::fill example
2  #include <iostream>
3  #include <array>
4
5  int main () {
6    std::array<int,6> myarray;
7
8    myarray.fill(5);
9
10   std::cout << "myarray contains:";
11   for ( int& x : myarray) { std::cout << ' ' << x; }
12
13   std::cout << '\n';
14
15   return 0;
16 }
```

Output:

```
myarray contains: 5 5 5 5 5 5
```

Linear: Performs as many assignment operations as the size of the array object.

Iterator validity

No changes.

Data races

All contained elements are modified.

Versions matter!

```
113       // DR 776.
114       _GLIBCXX20_CONSTEXPR void
115       fill(const value_type& __u)
116       { std::fill_n(begin(), size(), __u); }
```

https://gcc.gnu.org/onlinedocs/gcc-13.2.0/libstdc++/api/a00047_source.html

```
1102    template<typename _OutputIterator, typename _Size, typename _Tp>
1103      _GLIBCXX20_CONSTEXPR
1104      inline _OutputIterator
1105      __fill_n_a(_OutputIterator __first, _Size __n, const _Tp& __value,
1106                 std::input_iterator_tag)
1107      {
1108  #if __cplusplus >= 201103L
1109        static_assert(is_integral<_Size>{}, "fill_n must pass integral size");
1110  #endif
1111        return __fill_n_a1(__first, __n, __value);
1112      }

1071    template<typename _OutputIterator, typename _Size, typename _Tp>
1072      _GLIBCXX20_CONSTEXPR
1073      inline typename
1074      __gnu_cxx::__enable_if<__is_scalar<_Tp>::__value, _OutputIterator>::__type
1075      __fill_n_a1(_OutputIterator __first, _Size __n, const _Tp& __value)
1076      {
1077        const _Tp __tmp = __value;
1078        for (; __n > 0; --__n, (void) ++__first)
1079          *__first = __tmp;
1080        return __first;
1081      }
```

https://gcc.gnu.org/onlinedocs/gcc-13.2.0/libstdc++/api/a00695_source.html

https://cplusplus.com/reference/array/array/fill/

public member function

std::vector::erase

<vector>

C++98 C++11

```
iterator erase (iterator position);iterator erase (iterator first, iterator last);
```

Erase elements

Removes from the vector either a single element (***position***) or a range of elements (`[first,last)`).

This effectively reduces the container size by the number of elements removed, which are destroyed.

Because vectors use an array as their underlying storage, erasing elements in positions other than the vector end causes the container to relocate all the elements after the segment erased to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as list or forward_list).

Parameters

`position`

Iterator pointing to a single element to be removed from the vector.
Member types `iterator` and `const_iterator` are random access iterator types that point to elements.

`first, last`

Iterators specifying a range within the vector] to be removed: `[first,last)`. i.e., the range includes all the elements between ***first*** and ***last***, including the element pointed by ***first*** but not the one pointed by ***last***.
Member types `iterator` and `const_iterator` are random access iterator types that point to elements.

Return value

An iterator pointing to the new location of the element that followed the last element erased by the function call. This is the container end if the operation erased the last element in the sequence.

Member type `iterator` is a random access iterator type that points to elements.

Complexity

Linear on the number of elements erased (destructions) plus the number of elements after the last element deleted (moving).

Iterator validity

Iterators, pointers and references pointing to ***position*** (or ***first***) and beyond are invalidated, with all iterators, pointers and references to elements before ***position*** (or ***first***) are guaranteed to keep referring to the same elements they were referring to before the call.

Data races

The container is modified.
None of the elements before ***position*** (or ***first***) is accessed, and concurrently accessing or modifying them is safe.

https://cplusplus.com/reference/vector/vector/erase/

Variable Size **List** “Container” in STL

- Backed by a native array
- Size grows or shrinks, typically logarithmically
- Number of items can be dynamic

https://cplusplus.com/reference/list/list/

Linked List Representation

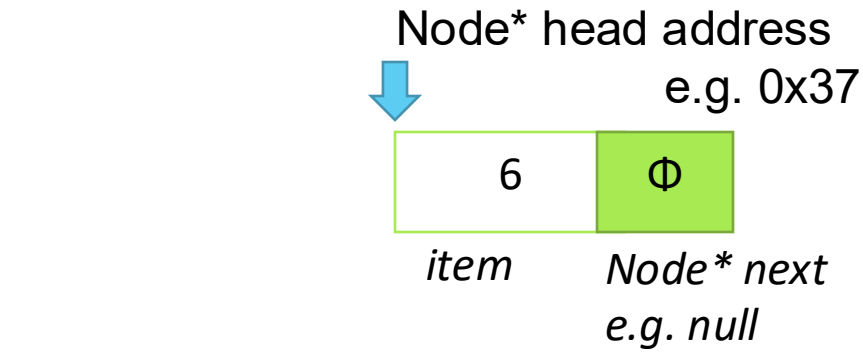
- **Problems with array**
  - Pre-defined capacity, under-usage, cost to move items when full. Fixed-size items (primitives)
- **Solution:** Grow data structure dynamically when we add or remove ==> Only use as much memory as required
- *Linked lists* use ***pointers*** to contiguous chain items
  - **Node** structure contains **item** and pointer to **next** node in List
  - Add or remove nodes when setting or getting items

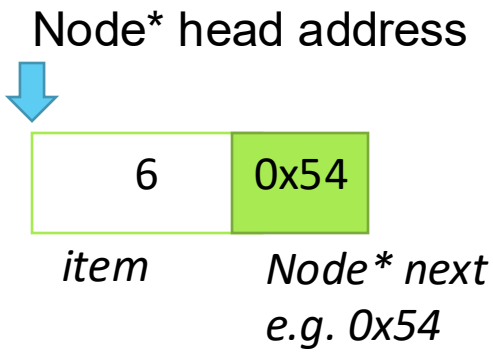
Try yourself!

Print the values of **pointer locations** for ***vector*** and linked ***list*** items. Is there any pattern?

Node & Chain

```
class Node {
        int item
        Node* next
}
class LinkedList {
        Node* head
        int size
        append() {...}
        get() {...}
        set() {...}
        remove {...}
}
```



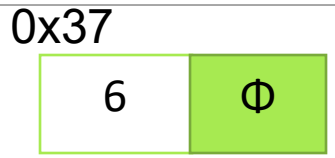


Linked List Operations

head=null

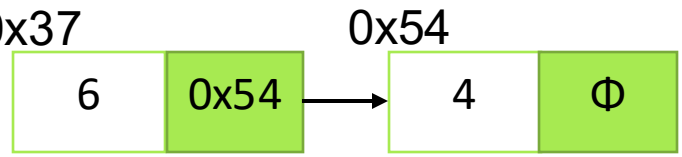
Initial empty list

Add item 6



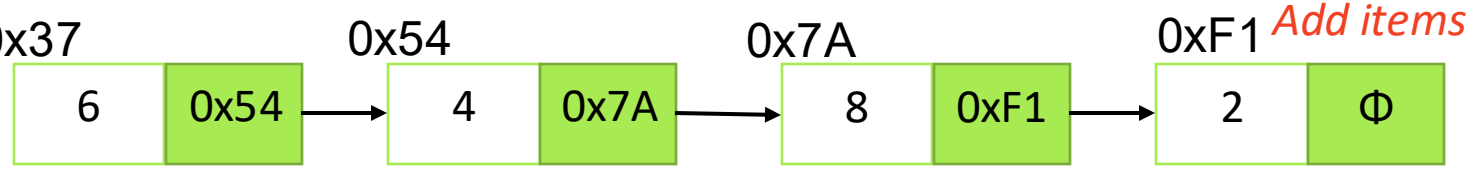
head=0x37

Add item 4



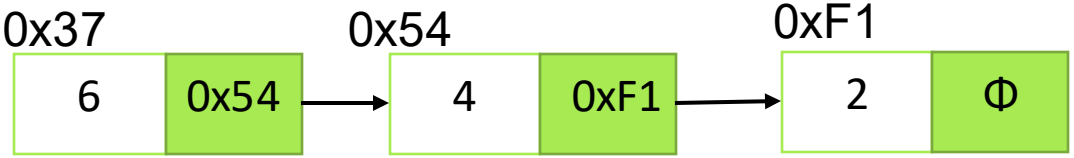
head=0x37

Add items 8, 2



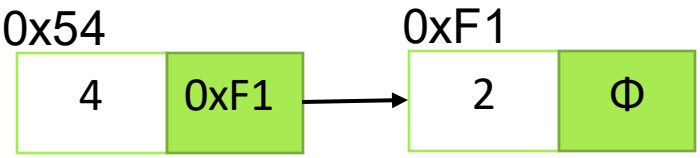
head=0x37

Remove 8



head=0x37

Remove 6



head=0x54

Live demo on VS Code

- Suppose I need a container to store numbers
- In my application, I desire fast appending of new elements
- Should I pick any arbitrary container in C++ that supports this feature?

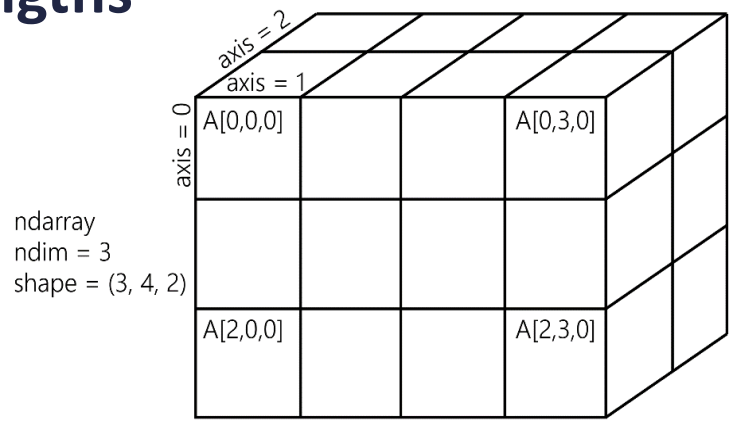
https://github.com/cjain7/DS221-Chirag-LiveDemos/tree/main/Lecture_1

Matrices & n-D Arrays

Matrices & n-D Arrays

- Arrays can have more than 1-dimension
  - 2-D Arrays are called **matrices,** higher dimensions called **tensors**
- Arrays have as many **indexes** for access as dimensions
  - A[i], B[i][j], C[i][j][k]
- Dimensions may have **different lengths**

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]



- Mapping from **n-D to 1-D array**
  - Items **n** dimensions "flattened" into **1** dimension
  - Contiguous memory locations in 1-D
  - Native support in programming languages

https://www.tutorialspoint.com/cprogramming/c_multi_dimensional_arrays.htm
https://ipython-books.github.io/13-introducing-the-multidimensional-array-in-numpy-for-fast-array-computations/

n-D Arrays as 1-D Arrays

- Memory locations are contiguous, like 1-D arrays
  - So multi-dimension arrays needs to be converted to a 1-D layout for constant time access
- Convert **A[i][j]** to **M[k]** … i=row index, j=column index, C=number of cols, R=number of rows
  - **Row Major Order** of indexing: k=map(i,j)=i*C+j
  - **Column Major Order** of indexing: k=map(i,j)=j*R+i

n-D Arrays as 1-D Arrays

- Memory locations are contiguous, like 1-D arrays
  - So multi-dimension arrays needs to be converted to a 1-D layout for constant time access
- Convert **A[i][j]** to **M[k]** … i=row index, j=column index, C=number of cols, R=number of rows
  - **Row Major Order** of indexing: k=map(i,j)=i*C+j
  - **Column Major Order** of indexing: k=map(i,j)=j*R+i
- *How does this look in memory location layout?*
- *How can you extend this to higher dimensions (tensors)?*

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

(a) Row-major mapping

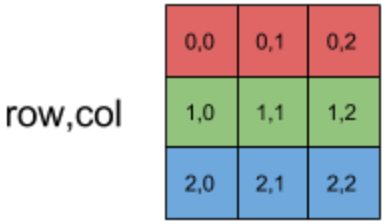
0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

(b) Column-major mapping

Figure 7.2 Mapping a two-dimensional array

Sahni Textbook, Chapter 7

Memory Layout

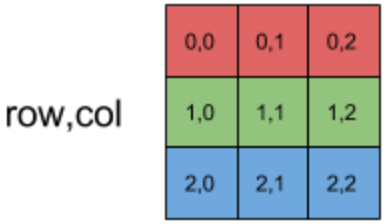


Row Major layout of 2D Array [R][C]

```
item_addr = base_addr +
size_of_item*(i*C + j)
```

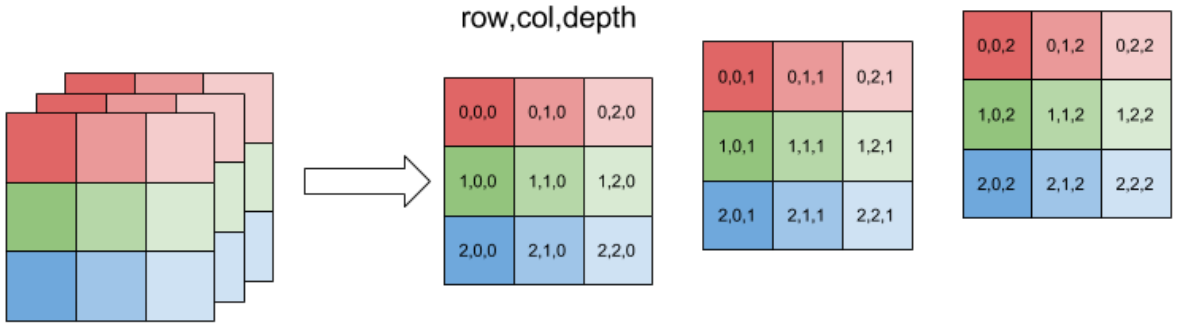
https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Memory Layout



Row Major layout of 2D Array [R][C]

```
item_addr = base_addr +
size_of_item*(i*C + j)
```



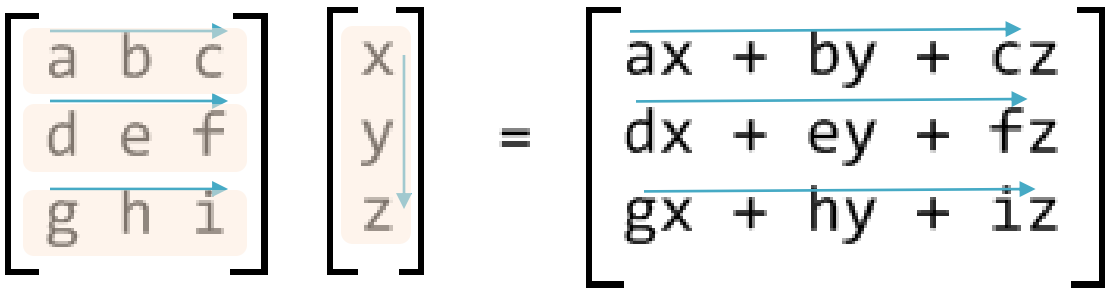
Row Major layout of 3D Array [R][C][D]

https://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays

Matrix-Vector Multiplication

```
// Given a[n][n], b[n][n]
// c[n][n] initialized to 0
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
  c[i] += a[i][j] * b[j];
```

Caching has an effect based on order of access!

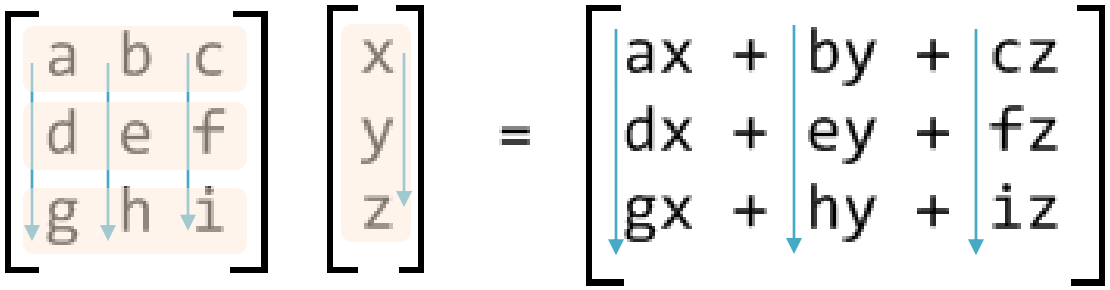


Matrix-Vector Multiplication

```
// Given a[n][n], b[n][n]
// c[n][n] initialized to 0
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    c[i] += a[i][j] * b[j];
```

Try yourself!

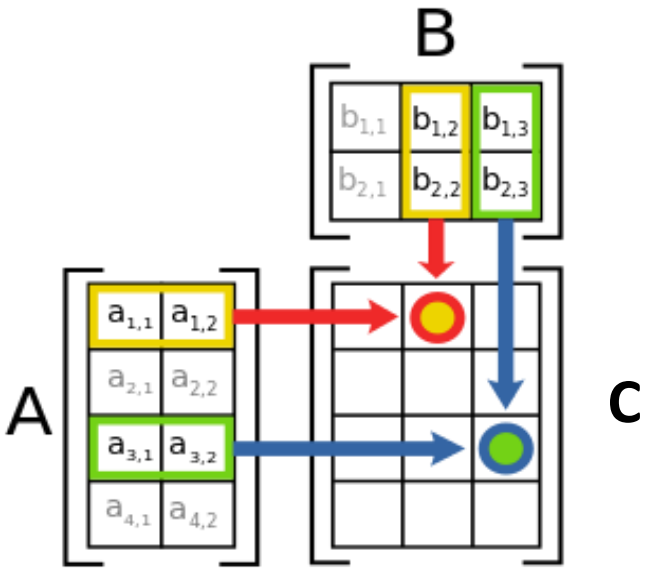
Write optimised codes for matrix-vector multiplication with: (a) matrix stored as 1D array in row-major order, and (b) with column-major order



Hint: Reading memory in contiguous locations is significantly faster than jumping around among locations.

Matrix Multiplication

```
// Given a[n][n], b[n][n]
// c[n][n] initialized to 0
for (i = 0; i < N; i++)
 for (j = 0; j < N; j++)
   for (k = 0; k < N; k++)
     c[i][j] += a[i][k] * b[k][j];
```



Does this exploit cache locality? Can we improve this?

https://en.wikipedia.org/wiki/Matrix_multiplication
https://passlab.github.io/CSCE513/notes/lecture10_LocalityMM.pdf
https://medium.com/@Styp/why-loops-do-matter-a-story-of-matrix-multiplication-cache-access-patterns-and-java-859111845c25

Competitions to write fast code!

Example: https://dmoj.ca/contest/fastmatrixmul (2019)

Problems

Problem
Fast Matrix Multiplication (1024x147 by 147x64)
Fast Matrix Multiplication (64x576 by 576x256)
Fast Matrix Multiplication (16x2304 by 2304x256)

Show organizations

Rank	Organization	Username	1 1000000	2 1000000	3 1000000	Points
1	Georgia Tech	xyz2606	563901.8	751452.7	600572.1	1915926
2	Colonel By	Zeyu	561146.8	724672.4	588467.9	1874287
3		Karshilov	537889.4	725378.3	596248	1859515
4		nhho	550218.5	700530	553669	1804417
5	Lord Byng	Dingledooper	547074.8	735937.2	470178.9	1753190

Live demo on VS Code

- Just for fun, let us compare how much time a naive $O(N^3)$ matrix multiple implementation takes compared to a well-optimized open-source implementation

https://github.com/cjain7/DS221-Chirag-LiveDemos/tree/main/Lecture_2

Sparse Matrices

- Only a small subset of items are populated in matrix
  - Examples: (1) Students and courses taken, (2) Faculty and courses taught
  - Adjacency matrix of social network graph
    - vertices are people, edges are “friends”
    - Rows and columns are people, cell has 0/1 value
- Why not use regular 2-D matrix?

Space taken = M*N

Sparse Matrices as 2-D arrays

- Coordinate List (COO) Format
- Each non-zero item has one entry in linear list
  - What type of items are in my list? Tuple of 3 integers <row, column, value>

Sparse Matrices as 2-D arrays

- Coordinate List (COO) Format
- Each non-zero item has one entry in linear list
  - What type of items are in my list? Tuple of 3 integers <row, column, value>
  - What is the index of each item? i^{th} non-zero item in the row-major order is at i^{th} position in the linear list

0	0	0	2	0	0	1	0	
0	6	0	0	7	0	0	3	
0	0	0	9	0	8	0	0	
0	4	5	0	0	0	0	0	

(a) A 4 × 8 matrix

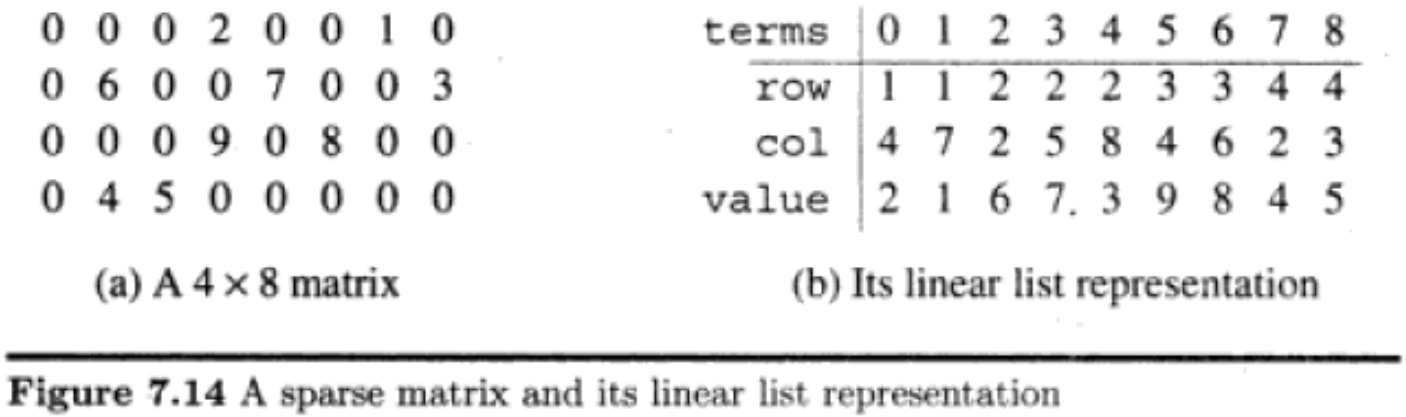
terms	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

(b) Its linear list representation

Figure 7.14 A sparse matrix and its linear list representation

Sparse Matrices as 2-D arrays

- Coordinate List (COO) Format
- Each non-zero item has one entry in linear list
  - What type of items are in my list? Tuple of 3 integers <row, column, value>
  - What is the index of each item? i^{th} non-zero item in the row-major order is at position i in the linear list



0	0	0	2	0	0	1	0	
0	6	0	0	7	0	0	3	
0	0	0	9	0	8	0	0	
0	4	5	0	0	0	0	0	

(a) A 4 × 8 matrix

terms	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

(b) Its linear list representation

Figure 7.14 A sparse matrix and its linear list representation

Space taken = 3*nnz (nnz = number of non-zero)

Sahni Textbook, Chapter 7

Sparse Matrix Addition

```
// Col is no. of cols in orig. matrix
// A, B are input sparse matrix. C is output sparse matrix.
while(p < pMax && q < qMax) {
   p1 = A[p].r*Col + A[p].c   // get index for A in orig. matrix
   q1 = B[q].r*Col + B[q].c
```

```
}
```

See Sahni, Program 7.17

Sparse Matrix Addition

```
// Col is no. of cols in orig. matrix
// A, B are input sparse matrix. C is output sparse matrix.
while(p < pMax && q < qMax) {
   p1 = A[p].r*Col + A[p].c   // get index for A in orig. matrix
   q1 = B[q].r*Col + B[q].c
   if(p1 < q1)                // Only A has that index
```

```
}
```

See Sahni, Program 7.17

Sparse Matrix Addition

```
// Col is no. of cols in orig. matrix
// A, B are input sparse matrix. C is output sparse matrix.
while(p < pMax && q < qMax) {
   p1 = A[p].r*Col + A[p].c  // get index for A in orig. matrix
   q1 = B[q].r*Col + B[q].c
   if(p1 < q1)                // Only A has that index
      C[k] = {A[p].r, A[p].c, A[p].val}          // Set entry
      p++
}
```

See Sahni, Program 7.17

Sparse Matrix Addition

```
// Col is no. of cols in orig. matrix
// A, B are input sparse matrix. C is output sparse matrix.
while(p < pMax && q < qMax) {
   p1 = A[p].r*Col + A[p].c  // get index for A in orig. matrix
   q1 = B[q].r*Col + B[q].c
   if(p1 < q1)                // Only A has that index
      C[k] = {A[p].r, A[p].c, A[p].val}          // Set entry
      p++
   else if(p1==q1)                    // Both A & B have that index
      C[k] = {A[p].r, A[p].c, A[p].val+B[q].val}   // Add vals
      p++
      q++



}
```

See Sahni, Program 7.17

Sparse Matrix Addition

```
// Col is no. of cols in orig. matrix
// A, B are input sparse matrix. C is output sparse matrix.
while(p < pMax && q < qMax) {
   p1 = A[p].r*Col + A[p].c  // get index for A in orig. matrix
   q1 = B[q].r*Col + B[q].c
   if(p1 < q1)                    // Only A has that index
      C[k] = {A[p].r, A[p].c, A[p].val}          // Set entry
      p++
   else if(p1==q1)                    // Both A & B have that index
      C[k] = {A[p].r, A[p].c, A[p].val+B[q].val}   // Add vals
      p++
      q++
   else                           // Only B has that index
      C[k] = {B[q].r, B[q].c, B[q].val}          // Set entry
      q++
   k++
}
```

See Sahni, Program 7.17

Compressed Sparse Row (CSR)

- Similar to 2-D array, but more **space efficient**
  - **V[nnz]**
  - **C[nnz]**
  - **R[M+1]**

Space taken = 2*nnz + (M + 1)

https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)
https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/

Compressed Sparse Row (CSR)

- Similar to 2-D array, but more **space efficient**
  - **V[nnz]** non-zero values in row-major order
  - **C[nnz]** column index of nnz in A … *Column offset* within the row group for a non-zero value
  - **R[M+1]**

Space taken = 2*nnz + (M + 1)

https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)
https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/

Compressed Sparse Row (CSR)

- Similar to 2-D array, but more **space efficient**
  - **V[nnz]** non-zero values in row-major order
  - **C[nnz]** column index of nnz in A … *Column offset* within the row group for a non-zero value
  - **R[M+1]** stores cumulative count of non-zero values till $(i-1)^{th}$ row … Offset to the start of nnz values for the i^{th} row in array A

Space taken = 2*nnz + (M + 1)

https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)
https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/

Compressed Sparse Row (CSR)

- Similar to 2-D array, but more **space efficient**
  - **V[nnz]** non-zero values in row-major order
  - **C[nnz]** column index of nnz in A … *Column offset* within the row group for a non-zero value
  - **R[M+1]** stores cumulative count of non-zero values till $(i-1)^{th}$ row … Offset to the start of nnz values for the i^{th} row in array A
  - i^{th} row elements are present from **V[R[i]]**
  - Existence of j^{th} col elements in **C**

Space taken = 2*nnz + (M + 1)

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

V = [5 8 3 6]nnz *Values (Row groups)*

C = [0 1 2 1]nnz *Column offset*

R = [0 0 2 3 4]M+1 *Offset to row group*

https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)
https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/

Compressed Sparse Row (CSR)

- **V[nnz]** non-zero values in row-major order
- **C[nnz]** column index of nnz in A … *Column offset* within the row group for a non-zero value
- **R[m+1]** stores cumulative count of non-zero values till $(i-1)^{th}$ row … Offset to the start of nnz values for the i^{th} row in array A

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

V = [5 8 3 6]nnz — *Values (Row groups)*

C = [0 1 2 1]nnz — *Column offset*

R = [0 0 2 3 4]m+1 — *Offset to row group*

To access X[i,j]:

- *Row i* values must be present between **V[s]** and until **V[e]**, where **s= R[i]** and **e= R[i+1]-1**
- Check corresponding column offset, **C[s]** until **C[e]**. If any of these C values match *j*, the value is present in the corresponding index of V.
- If **(e < s)** OR no C matches, then the value is 0

Try yourself!
Matrix-matrix addition using CSR

https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)
https://www.geeksforgeeks.org/sparse-matrix-representations-set-3-csr/

Tasks

- Self study (Sahni Textbook)
  - Chapters 5 & 6 “Linear Lists—Array & Linked Representations”
  - Chapter 7, Arrays and Matrices
- Programming Self Study
  - Try out **vector** and **list** data structures in **C++ STL**
  - Try out **matrix-matrix multiplication, matrix-vector multiplication** in C++
  - Try out **GitHub Copilot** on Visual Code