Indian Institute of Science
Bangalore, India
भारतीय विज्ञान संस्थान
बंगलौर, भारत

DS286 | 2016-08-24 & 26

L4: Linear Lists

Yogesh Simmhan

simmhan@cds.iisc.ac.in

CDS
The Department of Computational and Data Science

Primitive data types

- Boolean
- Integer
- Float
- Byte
- Character
- String
- Typically, are “atomic” data types
- Have well defined meanings and operations
  - Additions/subtraction, control flow tests, bit-shifting, length
- Common across different programming languages

Collections of data

- Data Structures to store collections of primitive data types
  - Primitive types are called `items, elements, instances, values`…depending on context
- Properties of the collection
  - Invariants that must be maintained, irrespective of operations
- Operations on the collection
  - Standard operations to create, modify, access elements
- Different **implementations** for same **abstract** collection

Linear List

- Properties
  - **Ordered** list of items…precedes, succeeds; first, last
  - **Index** for each item…lookup or address item by index value
  - **Finite Length** for the list…can be empty, size can vary
  - Items of **same type** present in the list
- Operations
  - Create, destroy
  - Lookup by index, item value
  - Find size, if empty
  - Add, delete item

1-D Array Representation

- Implementation of the abstract list data structure using programming language
  - "Backing" Data Structure
- **`arrays`** are **contiguous** memory locations with **fixed capacity**
- Allow elements of same type to be present at specific **positions** in the array
- **Index** in a **List** can be mapped to a **Position** in the **Array**
  - **Mapping function** from list index to array position

Mapping Function

- Say n is the capacity of the array
- Simple mapping
  - `position(index) = index`
- Reverse mapping
  - `position(index) = n - index - 1`
- Wrap-around mapping
  - `position(index) = (position(0)+index) % n`
  - `position(0) = x`

E.g. using same "backing" array for forward and reverse list.

E.g. using "backing" array for queues...add from front, remove from back.

List Operations

- `void set(index, item)`
- `item get(index)`
- `void append(item)`
- `void remove(index)`
- `int size()`
- `int capacity()`
- `boolean isEmpty()`
- `int indexOf(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 List {        // list with index starting at 1
    int arr[]       // backing array for list
    int capacity    // current capacity of array
    int size        // current occupied size of 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
     arrNue = int[MAX(index, 2*capacity)]
     // copy all items from old array to new
     // source, target, src start, trgt start, length
     copyAll(arr, arrNue, 0, 0, capacity)
     capacity = MAX(index, 2*capacity) // update var.
     delete(arr) // free up memory
     arr = arrNue
  }
  if(index < 1) {
     cout << "Invalid index:" << index << "Expect >=1"
  } else {
     int pos = index - 1
     arr[pos] = item
     size++
  } // end if
} // end set()
} // end List
```

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 used to shrink space
  - Why?

List Operations using Arrays

- `void append(item)`
  - Insert after current "last" item…use **`size`**
  - Sanity checks?
- `void remove(index)`
  - Remove item at index
  - Sanity checks?
- `int indexOf(item)`
  - Get "first" index of item with given value
  - Sanity checks?

List Operations using Arrays

- `int size()`
- `int capacity()`
- `boolean isEmpty()`

List Operations using Arrays

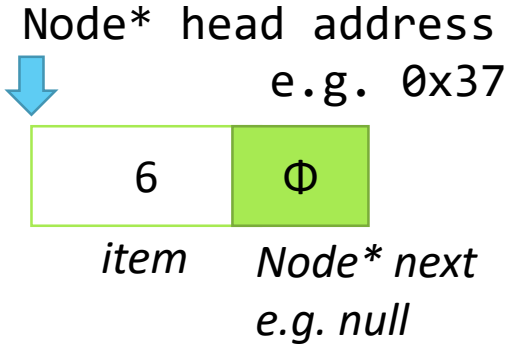
- Complexity
  - **Storage Complexity**: Amount of storage required by the data structure, relative to items stored
  - List using Array: …
  - **Computational Complexity**: Number of CPU cycles required to perform each data structure operation
  - size(), set(), get(), indexOf()
- Pros and Cons of using Arrays?

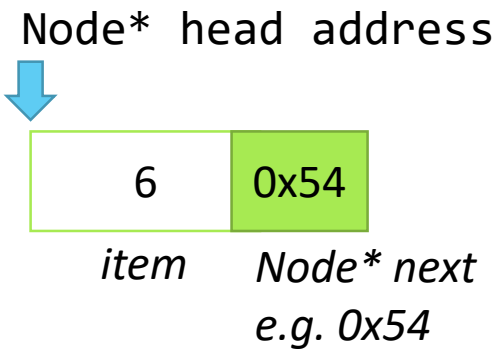
Linked List Representation

- Problem with array: Pre-defined capacity, under-usage, cost to move items when full
- Solution: Grow backing 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

Node & Chain

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



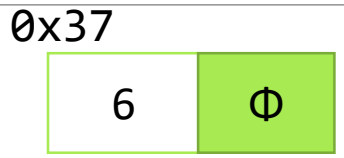


Linked List Operations

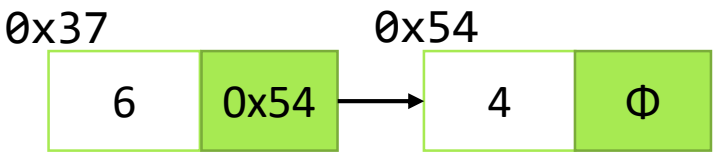
head=null

Initial empty list

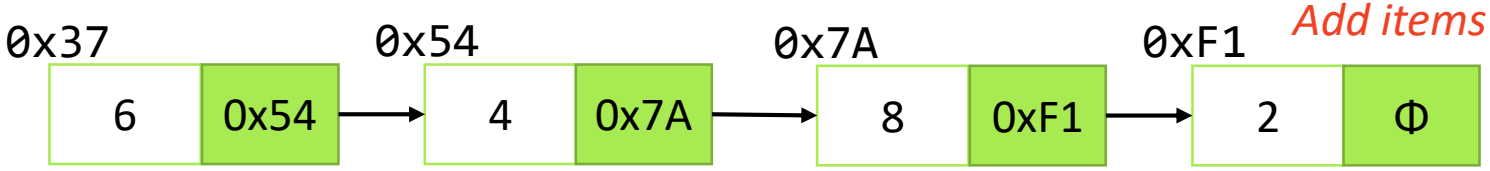
Add item 6



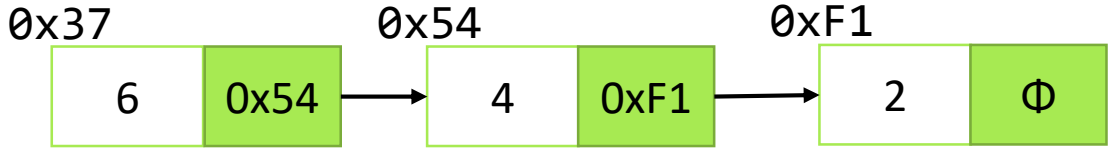
Add item 4



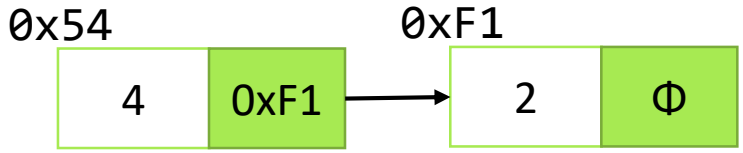
Add items 8, 2



Remove 3



Remove 1



Linked List Operations

- PseudoCode for
  - `get`
  - `set`
  - `remove`
- Other Operations
  - `set()`
  - `indexOf()`
  - `...`

Complexity

- Storage Complexity
  - Only store as many items as you need
  - But…
- Computational Complexity
  - set(), get(), remove()
  - indexOf()
- Other Pros & Cons?
  - Memory management, mixed item types

Choosing between List implementations

- When to pick array based List?
- When to pick Linked List?
- Other lists
  - Doubly linked list
  - Sequential lists & Iterators

n-D Arrays

- Arrays can have more than 1-dimension
  - 2-D Arrays are also called matrices
- Mapping from n-D to 1-D array
  - Convert A[i][j] to B[k] … i=row index, j=column index
  - Row Major Order of indexing: `k=map(i,j)=i*C+j`
  - Column Major Order of indexing: `k=map(i,j)=j*R+I`
- *Extend to 3+ dimension arrays?*

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

n-D Arrays

- Array of Arrays representation
- First find pointer for row array
- Then lookup value at column offset in row array
- *Pros & cons relative to using 1-D array representation?*

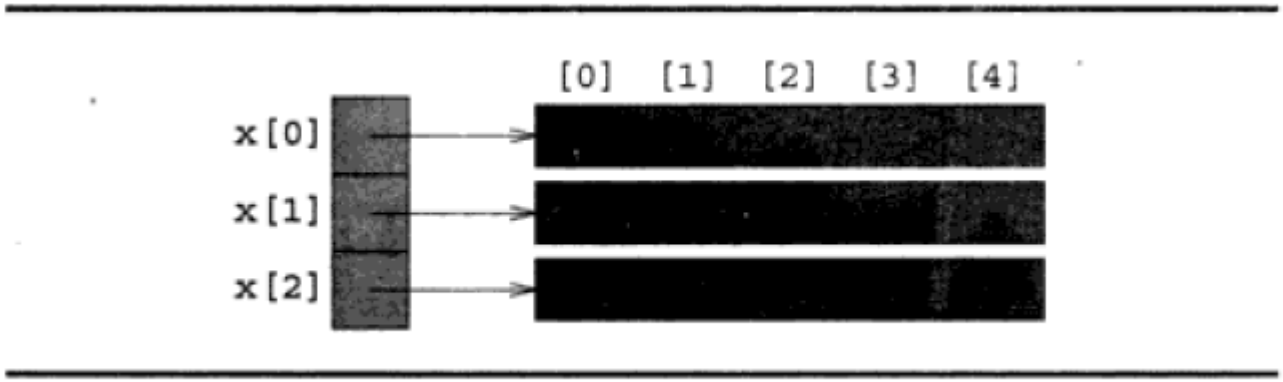
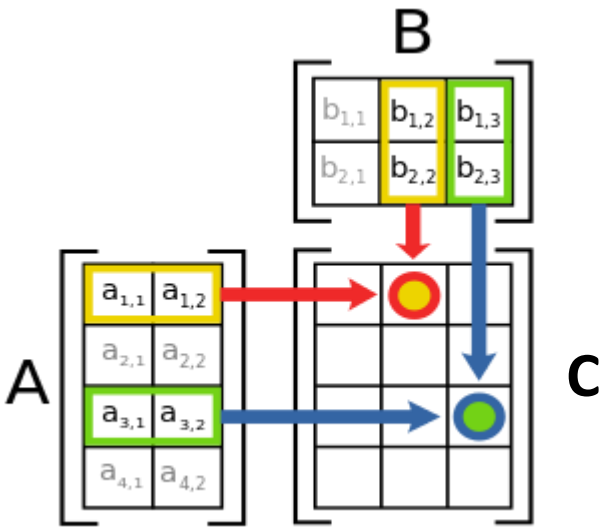


Figure 7.3 Memory structure for a two-dimensional array

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];
```



What is the time complexity?

https://en.wikipedia.org/wiki/Matrix_multiplication

Sparse Matrices

- Only a small subset of items are populated in matrix
  - Students and courses taken, faculty and courses taught
    - Product gives…
  - 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?
  - 1-D representation
  - Array of arrays representation

Sparse Matrices: Linear List representation

- Each non-zero item has one entry in list
  - index: <row, column, value>
  - index is the (i-1)th non-zero item in row-major order

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

Sahni Textbook, Chapter 7

Sparse Matrices: Addition

```
while(p < pMax && q < qMax) {
   p1 = A[p].r*C+A[p].c          // get single list index for A
   q1 = B[q].r*C+B[q].c
   if(p1<q1)                     // Only A has that index
      C[k] = <A[p].r, A[p].c, A[p].val>              // Copy
      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
      p++
      q++
   else                          // Only B has that index
      C[k] = <B[q].r, B[q].c, B[q].val>              // Copy
      q++
   k++
}
```


See Sahni, Program 7.17

Sparse Matrices: Linked List Representation

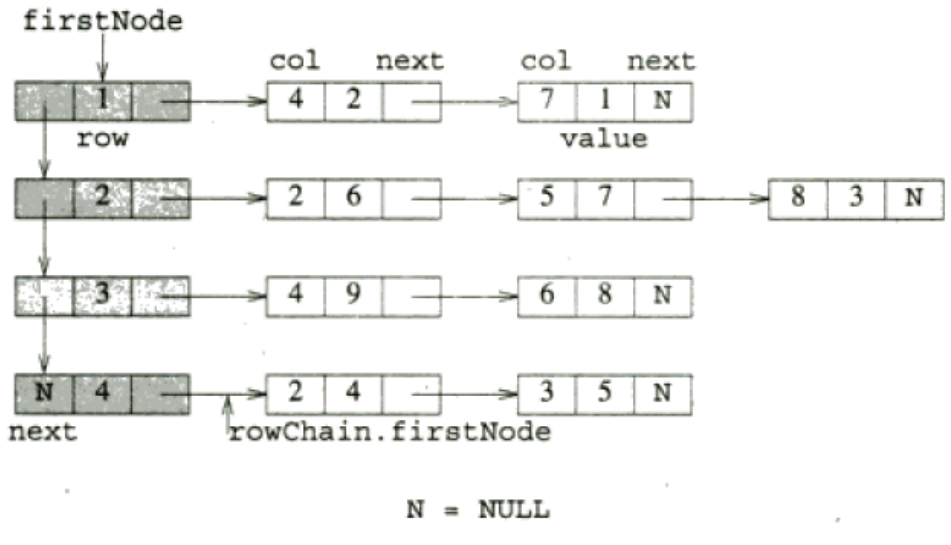
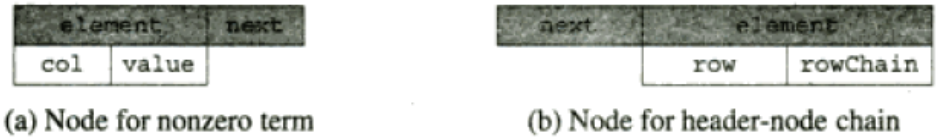


Figure 7.15 Linked representation of matrix of Figure 7.14(a)



(a) Node for nonzero term

(b) Node for header-node chain

Figure 7.16 Node structures in linked sparse matrix representation

See Sahni, Chapter 7.4

CodeChef (10% weightage)

- Send handle & profile link by 29 Aug, 2016 to mailing list
  - Everyone must list **IISc, Bangalore** as their affiliation
- Attempt test problem that will be posted on 29 Aug, 2016 before 31 Aug, 2016
- Start solving problems/coding contests independently using that handle between Sep 1 – Nov 30, 2016
- Submit list of successfully solved problems & contest ranks by Dec 1, 2016

Announcements & Tasks

- Self study (Sahni Textbook)
  - **Check**: Have you read Chapter 1 C++ Review, and Completed exercises 4, 6, 10, 13, 16, 19, testing of program 1.36?
  - **Read**: Chapters 5 & 6 "Linear Lists—Array & Linked Representations"
  - **Read**: Chapters 7.1, 7.4 "Arrays & Matrices"
  - **Read**: C++ primitive data types
  - **Read**: C++ Standard Template Library (STL) **`list`** and **`vector`** data structures
  - **Try:** Linked list implementation of List class from assignment
- Finish Assignment 1 by Sun Aug 28 *(50 points)*
  - All submissions MUST work (compile, run) on turing cluster!

Questions?