CMSC 420 – 0201 – Fall 2019
Lecture 02

Basic Data Structures

Fun Challenge – Arrays for Busy People

You are given a large integer n, and are asked to implement an array data structure $A[1, \ldots, n]$ of some type T, with the following operations:

- `init`(v): all elements of A are defined to be v
- `get`(i): return the value of $A[i]$, where $1 \leq i \leq n$
- `set`(i, x): set $A[i] = x$, where $1 \leq i \leq n$

Here is the catch … All the above operations must run in time $O(1)$, irrespective of the value of n. (Thus, you cannot use a loop to initialize the array.)

Rules:

1. You may use additional arrays, but you cannot assume they are initialized
2. No fancy data structures other than arrays are allowed
3. No bit manipulation is allowed (You cannot use a bit vector)

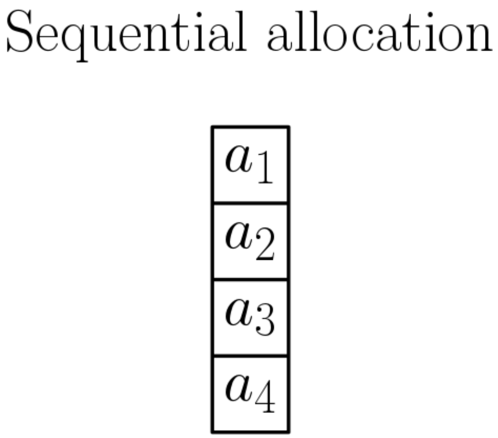
Linear Lists

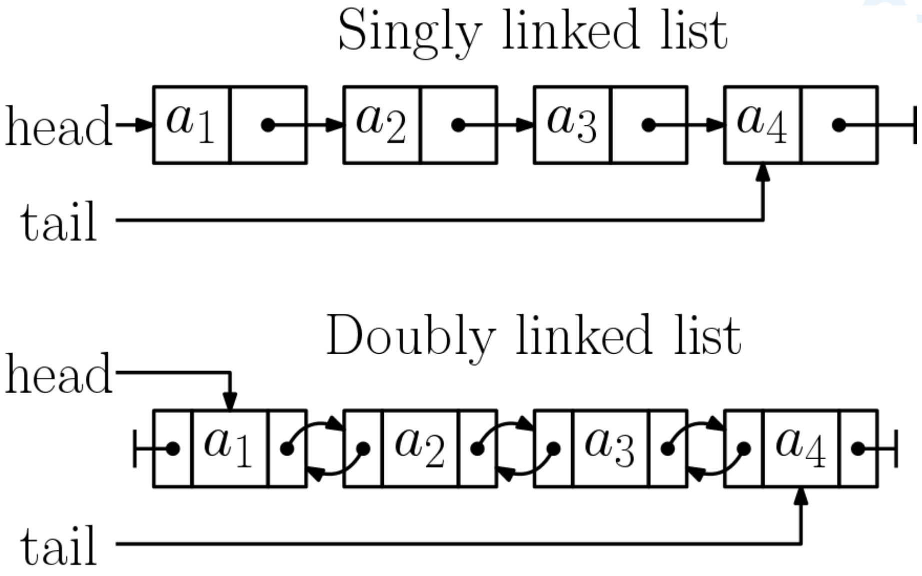
- Lists are among the most basic data types. Here is a simple interface.
  - `init()`: Initialize an empty list
  - `get(i)`: Returns element a_i
  - `set(i, x)`: Sets the ith element to x
  - `length()`: Returns the number of elements currently in the list
  - `insert(i, x)`: Insert element x just prior to element a_i (causing the index of all subsequent items to be increased by one)
  - `delete(i)`: Delete the ith element (causing the indices of all subsequent elements to be decreased by 1)

Linear Lists

Allocation Types

- Lists can be allocated in many ways. For example:
  - Sequential allocation - as an array
  - Singly linked - nodes, each referencing its successor
  - Doubly linked - nodes, each referencing its successor and predecessor





Linear Lists

Stacks, Queues, Deques

- There are a few very common types of lists:
  - Stacks - Supports insertion/removal from one end, called the top
    - push
    - pop
  - Queues - Supports insertion to tail end and removal from head end
    - enqueue
    - dequeue
  - Deque - Doubly-ended queue - Supports insertion/removal from either end
    - push-front, push-back
    - pop-front, pop-back
    - The name is a play on words, pronounced the same as "deck" of cards.

Linear Lists

Dynamic Storage Allocation

- When dealing with sequential allocation, what to do when we run out of space?
- Doubling:
  - Let n denote the current array size, and suppose we are asked to insert $(n + 1)$st item
  - Allocate a new array of size $2n$ (or generally any constant factor larger)
  - Copy the old contents to this array
  - Now, continue with the insertion
- Why double? Why not…
  - Less aggressive - Increase by a fixed number, say 100, more entries?
  - More aggressive - Increase by squaring the number of entries, say to n^2?

Linear Lists

Dynamic Storage Allocation

- Amortized cost: Given a sequence of m operations, the amortized cost of operations is the total cost of all the operations divided by the total number of operations, m.
- Theorem: When doubling reallocation is used, the amortized cost of stack, queue, and deque operations is O(1).
- Proof:
  - Charging argument - Each operation will perform a constant amount of work, and save a constant number of work tokens
  - When reallocation occurs, we will show that there are enough tokens to pay for the reallocation cost

Linear Lists

Dynamic Storage Allocation

- Theorem: When doubling reallocation is used, the amortized cost of stack operations is O(1).
- Proof:
  - Initialization - Assume a constant initial size O(1) initialization cost
  - Push - Do the operation, and deposit 4 work tokens in a bank account
  - Pop - Do the operation
  - Reallocation - Copy the current list of size n to a new array of size $2n$. We claim there are enough funds to pay for this. Why?
    - The last reallocation increased array size from $n/2$ to n
    - Since we overflowed, there must have been at least $n/2$ pushes since then
    - Bank account has at least $4(n/2) = 2n$. Thus, we have enough to pay for reallocation.

Multilists and Sparse Matrices

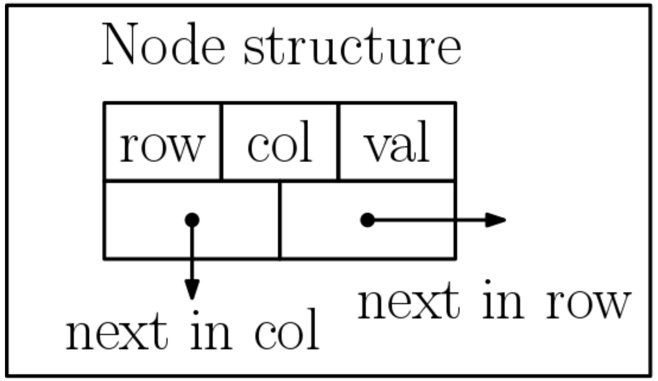
- Lists can be combined to perform more complex structures
- Example: Java’s ArrayList
- Better example: Sparse matrices
- Suppose you have a very large matrix, say $n \times m$, where n and m are in the tens of thousands.
- In many applications (particle dynamics in physics), almost all the matrix entries are zero

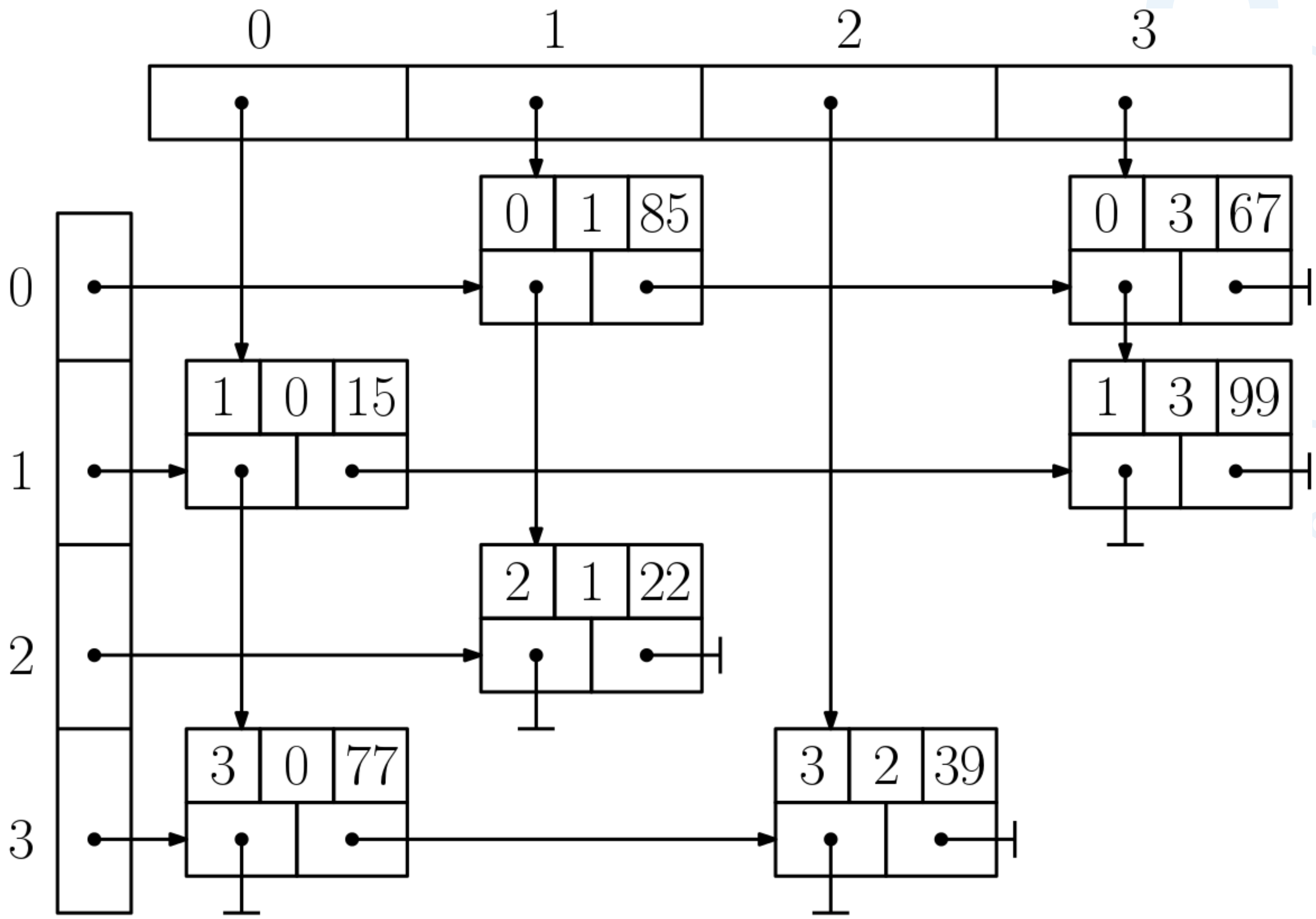
Multilist Representing a Sparse Matrix

Matrix contents

Column

Row	0	1	2	3
0	0	85	0	67
1	15	0	0	99
2	0	22	0	0
3	77	0	39	0





Fun Challenge – Arrays for Busy People

You are given a large integer n, and are asked to implement an array data structure $A[1, \ldots, n]$ of some type T, with the following operations:

init(v): all elements of A are defined to be v

get(i): return the value of $A[i]$, where $1 \leq i \leq n$

set(i, x): set $A[i] = x$, where $1 \leq i \leq n$

Here is the catch … All the above operations must run in time $O(1)$, irrespective of the value of n. (Thus, you cannot use a loop to initialize the array.)

Rules:

1. You may use additional arrays, but you cannot assume they are initialized
2. No fancy data structures other than arrays are allowed
3. No bit manipulation is allowed (You cannot use a bit vector)

Solution – Arrays for Busy People

We need to keep track of the entries of A that are defined, but how?

Idea 1: Maintain a parallel boolean array indicating which elements of A are defined: isDefined$[i]$ = true if $A[i]$ is defined.

$$\text{get}(i) := (\text{isDefined}[i] \;?\; A[i] : v)$$

Problem: We need to initialize this array, which will take $O(n)$ time. Too long!

Idea 2: Maintain a stack of indices of A that have been defined a value. (Can be initialized in constant time by setting $top = 0$.) We can "define" an entry $A[i]$ by pushing i on the stack.

Problem: Need to search the entire stack to test whether an entry is defined. Too long!

Solution – Arrays for Busy People

Here's the trick: Do both.

- Maintain a stack S containing the indices of defined elements.
- Maintain a parallel array $B[1, \ldots, n]$, where $B[i]$ indicates the element of the stack that witnesses that $A[i]$ is defined.
- Note that B may contain garbage, but the stack validates its "legitimate" entries.

	A		B		S	
1	cat	1	2	1	4	trusted
2	***	2	*	2	1	← top
3	***	3	*	3	*	
4	pig	4	1	4	*	
5	***	5	*	5	*	

set(5,"dog") →

	A		B		S	
1	cat	1	2	1	4	
2	***	2	*	2	1	trusted
3	***	3	*	3	5	← top
4	pig	4	1	4	*	
5	dog	5	3	5	*	

Solution – Arrays for Busy People

In addition to A, we maintain two arrays, $B[1, \ldots, n]$ and a stack S.

1. The command $\texttt{init}(v)$ saves the value of v and sets the stack empty ($top \leftarrow 0$).
2. When an entry $A[i]$ is first defined a value x, we push index i onto the stack, signaling that this entry has been initialized. We set $B[i] \leftarrow top$, which validates this entry. (Note that, $1 \leq B[i] \leq top$ and $S[B[i]] = i$.) Finally, we set $A[i] \leftarrow x$.
3. To test whether $A[i]$ is defined, test whether $1 \leq B[i] \leq top$ and $S[B[i]] = i$.
4. The command $\texttt{set}(i, x)$ applies Step 3 to test whether $A[i]$ is already defined. If not, we apply Step 2 to define it. If it was defined, we set $A[i] \leftarrow x$
5. The command get(x) applies Step 3 to test whether $A[i]$ is defined. If so, it returns $A[i]$. Otherwise it returns the default value v.

Do you believe it? You should be skeptical. Try it on a few examples to convince yourself.

Summary

- Basic data structures – Linear lists
- Stacks, queues, and deques
- Dynamic reallocation through doubling and amortized analysis
- Multilists
- (Fun problem – Not covered on the exams)