Hybrid Implementation

Alternate method to use arrays as storage
- Non-sequential
- Ordered and Efficient
- Simulates what the operating system does for dynamic allocation

Motivation

Motivational Problem: (from Homework set)

Suppose you have 200 statically allocated storage slots and wish to have two stacks

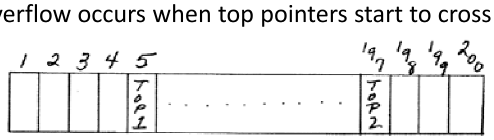
Classic solution: Divide space in half and use 100 slots for each stack in a traditional, array based implementation, as discussed in Module 4.

Breaks down if either stack > 100.

Motivation

Possible Solution:

•Use all 200 slots as a single array.
•Represent a stack at each end.
•Left stack is just like we have seen before.
•Right stack requires slight tweaking
•Overflow occurs when top pointers start to cross



Motivation

Pros:

- Each stack can > 100,
- Each stack can be ≤ 200
- They cannot both be large at the same time.

Cons:

- Two sets of basic stack methods (operations)
- What if you need a third stack?
- What if you want to manage a queue instead of a stack?

How can we generalize this solution?

4

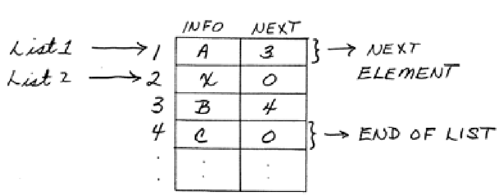
Hybrid Implementation

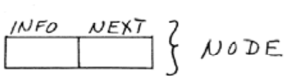
- Before the array indices determined the order of the data
- Here, the user determines the order
- We will have list nodes with a data and reference portion, like we did in dynamic allocation

- <u>Now,</u> we will declare an array of these nodes.

5

Hybrid Implementation





6

	1	26	0
	2	11	10
	3	5	16
List4 →	4	1	25
List2 →	5	17	1
	6	13	2
	7		
	8	19	19
	9	14	13
	10	4	22
	11		
List3 →	12	31	8
	13	6	3
	14		
	15		
	16	37	24
List1 →	17	3	21
	18		
	19	32	0
	20		
	21	7	9
	22	15	0
	23		
	24	12	0
	25	18	6

List1:
node[17].info = 3
[21] = 7
[9] = 14
[13] = 6
[3] = 5
[16] = 37
[24] = 12

List2:
node[5].info = 17
[1] = 26

List3:
node[12].info = 31
[8] = 19
[19] = 32

List4:
node[4].info = 1
[25] = 18
[6] = 13
[2] = 11
[10] = 4
[22] = 15

Note:
- Adjacent items on a list are not physically adjacent.
- Position in list does not correspond to position in array.
- Can have any number of lists
- Can have any type of list: General lists, queues, stacks, etc.
- Theses structures act like the dynamically allocated structures we have seen and have similar costs.
- Programmer is responsible for the housekeeping – not the system

7

	1	26	0
	2	11	10
	3	5	~~16~~ 24
	4	1	25
	5	17	1
	6	13	2
	7	90	0
	8	19	19
	9	14	13
	10	4	22
	11		
	12	31	8
	13	6	3
	14		
	15		
	16	37	24
List1 →	17	3	21
	18		
	19	32	0
	20		
	21	7	9
	22	15	0
	23		
	24	12	~~0~~ 7
	25	18	6

ADD 90 TO List1:
[17] 3
[21] 7
[14] 9
[13] 6
[3] 5
[16] 37
[24] 12
[7] 90

DELETE 37 at [16]:
[17] 3
[21] 7
[14] 9
[13] 6
[3] 5
[24] 12
[7] 90

→ storage becomes available!

8

	Info	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		11
11		12
12		-1

Initial State
- Each "node" refers to the next physically adjacent slot in the array
- Easy to do with a FOR Loop
- "-1" represents a NULL pointer

Management of Free Space
- Programmer writes methods
  - Getting free space
    - Easy modification of pop
  - Returning free space
    - Easy modification of push
- Manage it like a stack

9

Summary of Structures

List Type

General List
  Sorted
  Unsorted

Stack

Queue
  Priority Queue

Deque

Other Access Restricted
  Structures

Tree
  There are lots of
  variations on trees

Implementations

Array based
  Traditional
  Using Marked deletes

Linked (references)
  Headers
  Circular
  Doubly-linked
  Multiply-Linked

Hybrid

Methods

Insert
Delete
Search
Copy
Print
Peek
etc.

PRACTICE-Pick a reasonable cross section of combinations and write some methods.

What are the costs?

10