Elementary Data Structures: Linked Lists

Alexandra Stefan

What to Review

- Pointers – STUDENT SELF REVIEW (not reviewed in class)
  - “Pointers and Memory”: http://cslibrary.stanford.edu/102/PointersAndMemory.pdf
    - See more details in the next couple of slides.
    - Heap vs stack
  - See the pointers quiz posted on the Slides and Resources page.
- C struct and accessing individual fields with . and -> as appropriate
- typedef
- Linked lists: *concept* and *implementation.*
  - “Linked Lists Basics”: http://cslibrary.stanford.edu/103/LinkedListBasics.pdf
  - In exam, expected to write code involving lists.
- Read the reference material and the following slides.

Pay attention to the box representation showing what happens in the memory. You must be able to *produce such representations*.

- E.g. see page 13: think about what parts of the drawing are generated (or correspond) to each line of code.
- Recommended to read all the reference material. At the very least read and make sure you understand the parts highlighted in the review slides.

Good resources for review

If you did not do it already, solve the Pointer Quiz posted in the Code section.

From the Stanford CS Education Library:

- Pointers and Memory: http://cslibrary.stanford.edu/102/PointersAndMemory.pdf

  Skim through all and pay special attention to:

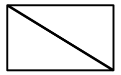
  - **Victim & TAB example on page 14**
  - Section 4 (Heap)
    - Examples on page 28:
      - middle example- draw the picture after each line in step 2, they show the final picture.
        - » Replace intPtr = malloc()sizeof(int));
        - » With: int * temp = malloc(sizeof(int));
        - » intPtr = temp; // DRAW pictures; Do we need to free intPtr and temp?
      - Notice the gray arrow in the third example: intPtr still points to a memory location – dangling pointer
    - Page 30: notice size+1 needed to store a string.

- Linked Lists Basics: http://cslibrary.stanford.edu/103/LinkedListBasics.pdf
  - Notice Heap Memory versus Stack Memory
  - **See WrongPush and CorrectPush examples (pages 12-15)**.

- Throughout our examples,
  - Think about what data is on the heap and what data is on the stack.
  - Use drawings to represent the memory and the variables in the program.

Pointer Review – by STUDENT

- Pointers and Memory:
  http://cslibrary.stanford.edu/102/PointersAndMemory.pdf

  Convention: NULL pointer: 

  Convention: bad pointer: $\boxed{XXX}$

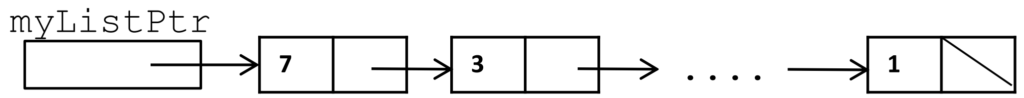
  Pointer, pointee, dereference [a pointer]

  Read all and pay special attention to:

  – Pointer representation and assignment: pages 3, 4
  – Examples in pages 7-8 (especially page 8)
  – **Victim & TAB example on page 14**
  – Section 4 (Heap Memory)
    - Examples on page 28:
      – middle example (draw the picture after each line, they show the final picture)
      – Notice the gray arrow in the third example: intPtr still point to a memory location– dangling pointer
    - Page 30: notice size+1 needed to store a string.

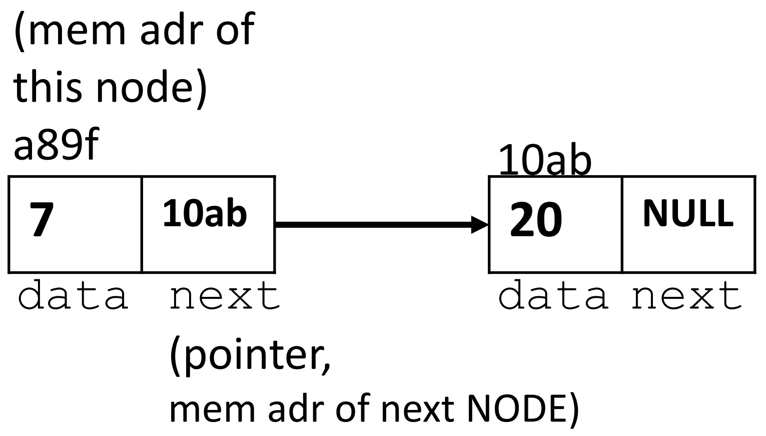
Linked List Review – The Concept

- Data structure that holds a collection of objects (kept in nodes).
- Why use it?: memory efficient & flexibility for insertion and deletion.
- Consists of a sequence of nodes.
- Each node stores an object (data) and a link to the next node. (*variations)
- The nodes are accessed by following the links (from one node to the next one).
- The list is accessed starting from the first node.
- The end is reached when the link to the next object is NULL.
- Add a new object:
  - Create a new node (allocate memory, draw the box),
  - Populate it (with the object and possibly set the link to NULL),
  - Update the links to connect it in the list.
- Delete an object
  - Use a temp variable to reference the node with the object that will be deleted.
  - Update the links in the list (so that it is skipped).
  - Destroy the node that was removed (free the memory, delete the box).

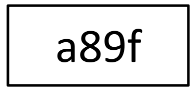


NODE vs POINTER to node

```
// struct for a node. drawing shows 2 boxes
struct node {
    int data;
    struct node * next;
}; // size: 4Bytes (int)+8Bytes(mem adr) = 12Bytes
```



```
typedef struct node * nodePT; /* size: 8Bytes
Another name for the type: struct node *
(can simply be used instead of struct node * ) */
```



Assume sizes:

char – 1 Byte

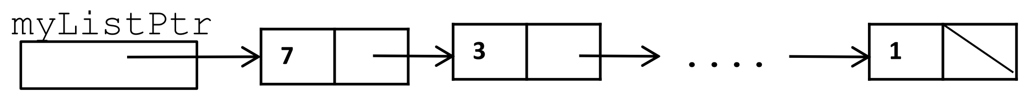
int – 4Bytes

double – 8 Bytes

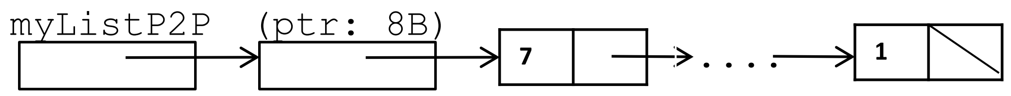
Memory address (pointer) – 8 Bytes

Representing a Linked List in C

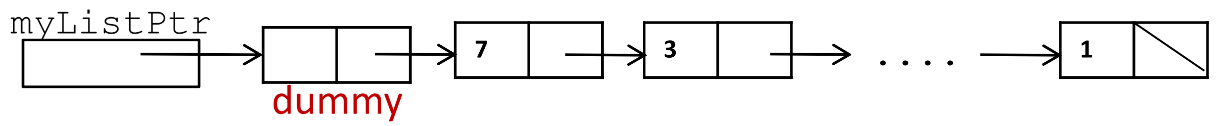
- Here: pointer=memory address
- As a POINTER to the FIRST NODE in the list. **Bad, but we will use this for simplicity**:
  - Lists have the same type as nodes.
  - It changes if the list is empty or a node is inserted (or deleted) at the beginning of the list. Conceptually, a variable representing a list should not have to change because we insert or delete a link at the beginning.



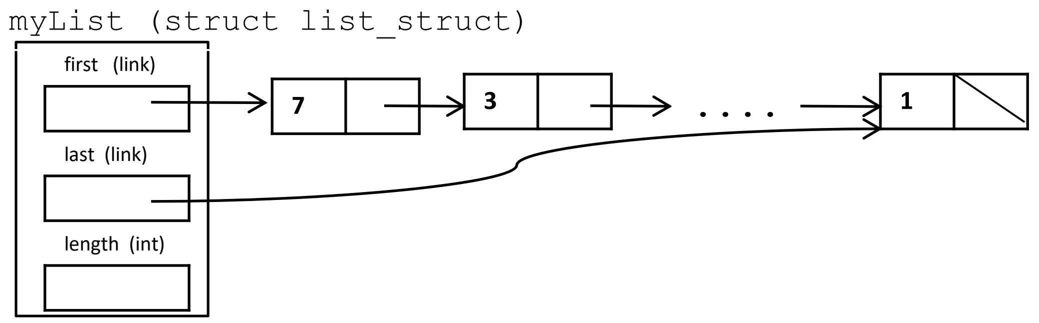
- As a POINTER to the POINTER to the first NODE
  - Advantage: once created, the list reference does not change.



- As a POINTER to a DUMMY NODE that will point to the first actual node of the list.
  - Solves some of the problems from above, but lists and nodes are still the same type.
  - The dummy node does not hold any useful data. The useful data starts in the first node after the dummy node



- ** As an new struct object that stores the pointer/link to the first actual node in the list and possibly some other data.
  - Better design.



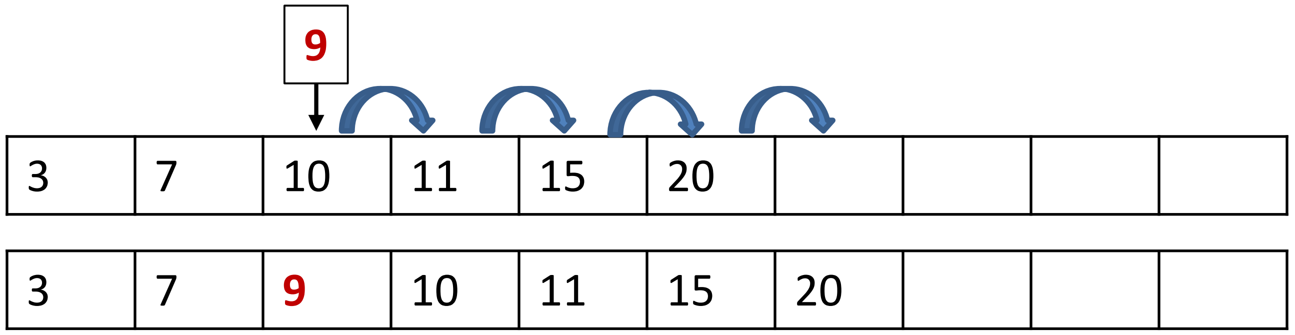
Summary: Linked Lists vs. Arrays

Operation	Arrays	Linked Lists
Access position *i*	***Θ****(1)*	***O****(N)* exact *Θ(i)* *(worst case: Θ(N))*
Modify position *i*	***Θ****(1)*	***O****(N)* exact *Θ(i)* *(worst case: Θ(N))*
Delete at position *i*	***O****(N)* exact *Θ(N-i)* *(worst case: Θ(N))*	***Θ****(1)* (assuming you have the previous node)
Insert at position *i*	***O****(N)* exact *Θ(N-i)* *(worst case: Θ(N))*	***Θ****(1)* (assuming you have the previous node)
When create it	*Must know max size*	*Do not need to know size.*
Space usage	*Fixed size (may waste space)*	*Grows & shrinks with the data.*

Here N is the number of items in the array or linked list.

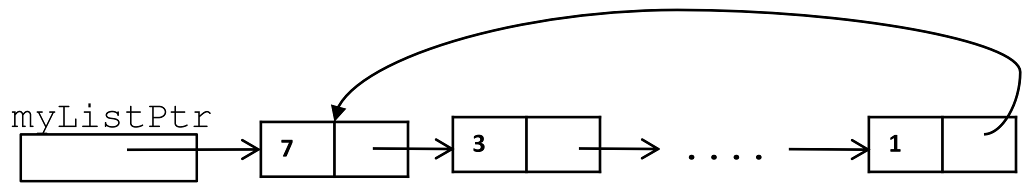
Position is the array cell for arrays and the node for linked lists.

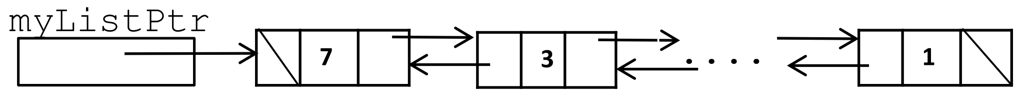
E.g. inserting 9 at index 2 requires copying to the right the cells at indexes (N-1) to 2 to make room for it.



Practice / Review

- Other types of linked lists
  - Circular list
    - Application: Round Robin selection in resource allocation
  - Double-linked list
    - Convenient access in any direction
    - Extra space needed





- Practice:
  - See wrongPush function in LinkedList document from Stanford
    - See correct push as well.
    - How will it be modified if the list has a dummy node?
  - Swap the first 2 nodes in a single-linked list. Code should not crash.
    - Extra: How do you do it in a double-linked list?

Practice from exam

Problem from Spring 19, exam 1
Answer both d and e before discussing it in class.
d) (8 pts) Assume the code below is correct. Draw boxes to show all the memory used by this program (for local variables and dynamically allocated). Write memory addresses and show the content of each box.

```
struct node * A;    //line 1
struct node * B = malloc(sizeof(struct node)); //line 2
A = B;          //line 3
B->next = A;    //line 4
```

e) (4 pts) Give the total memory usage (in bytes) for the program above (in part d). Assume sizes use din class and hw: char is 1B, int is 4B, double is 8B, memory address is 8B.

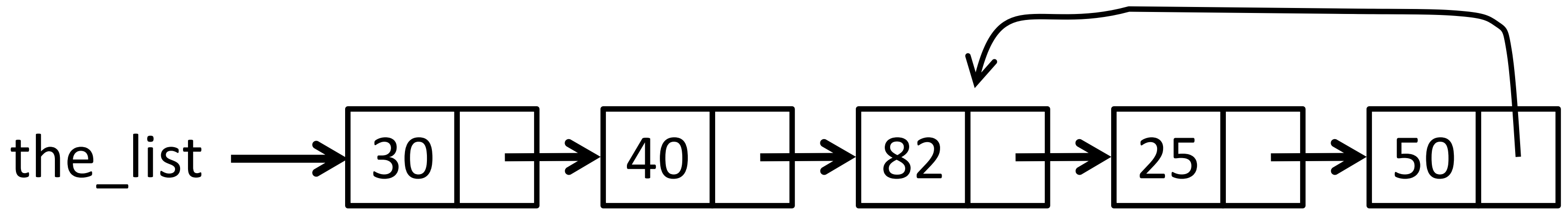
Interesting Problems

- Use **insertion sort** (or a variation of it) to sort a SINGLE linked list in place: move the nodes around. (Do not make a copy of it, do not move the contents of the nodes)
- Given a single-linked list, add another list that gives access to the nodes of the original list in the other direction (last to first).
  - Do NOT copy the data from the original list.
- **Indirect sort** a linked list, L. Indirect sort used a Data array and an array A of indexes. Your data here is the linked list, L, and for A, you can use an array or another linked list.
- Merge two sorted linked lists.
- Detect if there is **a cycle in a single-linked list**
  - See next page for more details

Detecting a Cycle in a Single Linked List

- Assume a list representation where you do NOT know the length of the list. E.g. the list is the pointer to the first node.

- Pb 1: Detect if a list has a cycle. No requirement of efficiency.
  - Have in mind that some initial items may not be part of the cycle:



- Pb 2: Detect if a list has a cycle **in *O(N)* time** (*N* is the number of unique nodes). (This is a good interview question)
  - Hint: You must use a geometric progression.

(Programming interviews: also practice expressing yourself clearly. Interviewers may ask a broad problem on purpose, expecting you to ask questions to narrow down it down: e.g. what happens in special cases such as when the list is empty.)

Why do I have to implement a Linked List in C?

1. Provides very good pointer understanding and manipulation practice (re-enforces your C knowledge).
2. This class is in C and after passing it, you are expected to be able to work with pointers and with linked lists in C.
3. It is one of the basic data structures.
    1. Granted, implementing it in other languages may not be as tricky as it is in C.
4. If you are familiar with it, you can easily adapt it as you need to (or even invent a new one).
5. It may end up being a programming interview question

   https://www.geeksforgeeks.org/top-10-algorithms-in-interview-questions/

See the slides on Linked Lists in C code review and **DRAWING of all the data used in that code.**

Even if you know Linked Lists, look at the slides to be able to draw them in detail.