

# Elementary Data Structures: Lists

CSE 2320 – Algorithms and Data Structures  
Based on presentation by Vassilis Athitsos  
University of Texas at Arlington

# What to Review

## – Pointers

- “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:

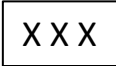
- **Victim & TAB example on page 14**
- Section 4 (Heap)
  - Examples on page 28:
    - middle example (draw the picture after each line, they show the final picture)
    - Notice the gray arrow in the third example: intPtr still points to a memory location – dangling pointer
  - Page 30: notice size+1 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

- Pointers and Memory:

<http://cslibrary.stanford.edu/102/PointersAndMemory.pdf>

Convention: NULL pointer: 

Convention: bad pointer: 

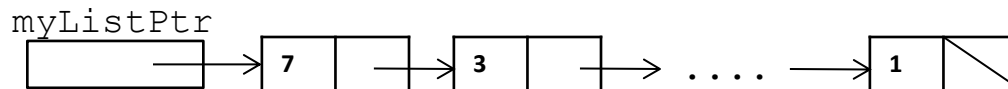
Pointer, pointee, dereference [a pointer]

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.

# Lists Review – The Concept (easy)

- Data structure that holds a collection of objects (kept in nodes).
- Advantages: memory efficient & flexibility for insertions and deletions.
- 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 (free the memory, delete the box).



# Links

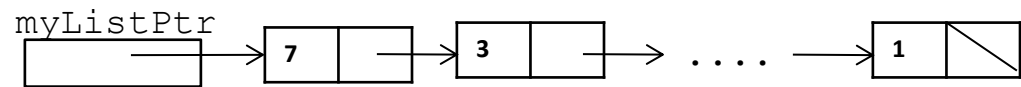
```
typedef int Item;
typedef struct node_struct * link;
struct node_struct {
    Item item;
    link next;
};
```

```
typedef struct list_struct * list;
struct list_struct {
    link first;
    link last;
    int length;
};
```

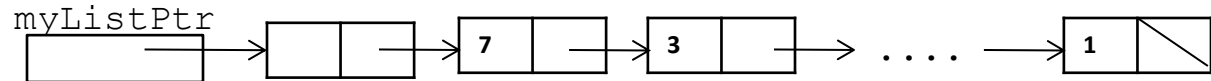
- Note: the item type can be **encapsulated using typedef**. It can be an int, float, char, or any other imaginable type.
- typedef simply renames the type. In this case:
  - int is renamed Item.
  - struct node\_struct \* is renamed link (and it is a pointer).
  - struct list\_struct \* is renamed list (and it is a pointer).

# Representing a List

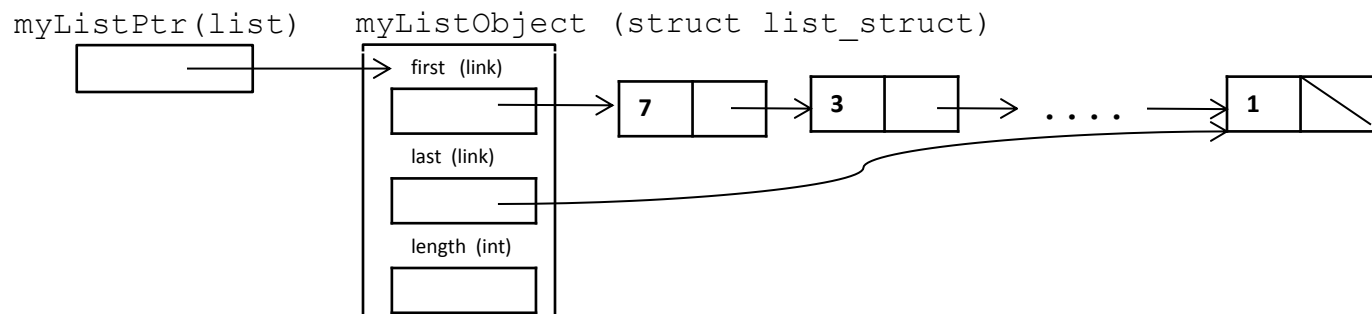
- How do we represent a list in code?
  - As **the first link**. Bad:
    - Lists have the same type as links.
    - Conceptually, a variable representing a list should not have to change because we insert or delete a link at the beginning.



- As **a dummy link** 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.



- As **an object (of a different type)** that stores the pointer/link to the first actual node in the list and possibly some other data.
  - Better design. We will use this version (with no last link).



# List Interface



# Abstracting the Interface

- When designing a new data type, it is important to **hide the details of the implementation** from the programmers who will use this data type (including ourselves).
- Why? So that, if we later decide to change the implementation of the data type, no other code needs to change besides the implementation.
- **List interface:**
  - A set of functions performing standard list operations:
    - Initialize/destroy a list.
    - Insert/delete a node (with a certain item).
    - Find a node (containing a certain item).
  - Every interaction with a list will be through these functions (NOT directly).

# List Interface

- The following files on the course website define and implement an abstract list interface:
  - list.h - provides the interface
  - list.c - implements this interface.
- Other code that wants to use lists can only see what is declared at list.h.
  - The **actual implementation of lists and nodes is hidden.**
- The implementation in list.c can change, without needing to change any other code.
  - For example, we can switch between our approach of lists and nodes as separate data types, and the textbook's approach of using a dummy first node.

# New compilation errors

- ‘incomplete type’
  - When try to use a member from a struct (defined in list.c) in the client code (i.e. read\_integers.c)
  - We want this behavior. This is why we have to use pointers to list\_struct instead of the struct itself. (What size does each one have?)
- “Multiple definitions of main”
  - The project contains two files that both have a ‘main’ function. Remove one of them.
- Numerous error messages
  - Missing semicolon in header file
- ‘..implicit cast to int..’ for a function call that returns a type other than int
  - That function may not have it’s header in the header file (list.h).
- Notice the usage of LIST\_H in the list.h file.
- Make a project in Netbeans, compile files on omega.

# Creating a new list

```
typedef struct list_struct * list;
```

```
struct list_struct
```

```
{  link first;
```

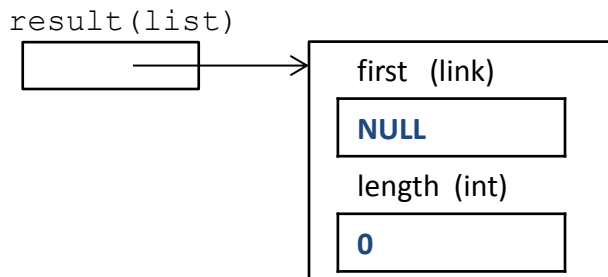
```
    int length;};
```

```
list newList()          // create an empty list
```

# Create an Empty List

```
typedef struct list_struct * list;
struct list_struct
{  link first;
    int length;};

list newList()          // create an empty list
{
    list result;
    result = (list)malloc(sizeof(*result));
    result->first = NULL;
    result->length = 0;
    return result;
}
```



Note:

- No nodes in the list.
- The list object is not NULL
- Convention: variable\_name (type)

# Destroying a List

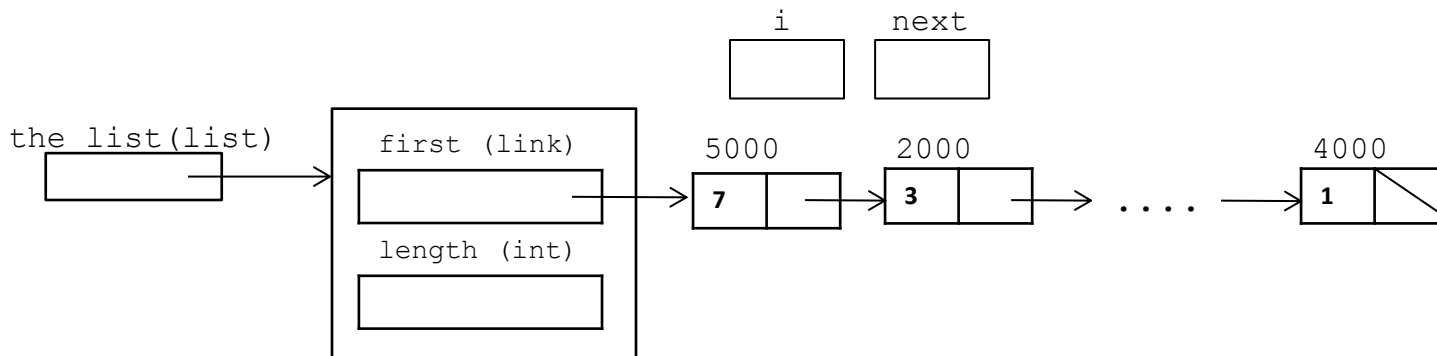
- How do we destroy a list?

```
void destroyList(list the_list)
```

Draw the list:

# Destroying a List

```
void destroyList(list the_list)
{
    link i = the_list->first;
    while(1)
    {
        if (i == NULL) break;
        link next = i->next;
        destroyLink(i); //uses the interface function, not free(i)
        i = next;
    }
    free(the_list); // very important!!!
}
```



# Inserting a Link

- How do we insert a link?

```
void insertLink(list my_list, link prev, link new_link)
```

- Assumptions:
  - We want to insert the new link right after link **prev**.
  - Link **prev** is provided as an argument.

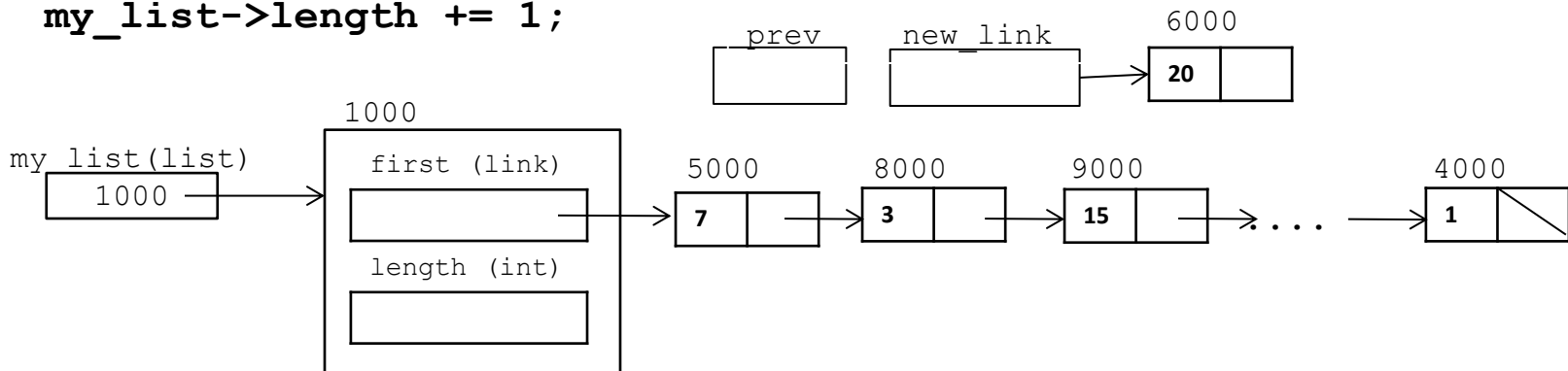


# Inserting a Link

// Inserts the new link right after link **prev** (provided as argument).

// Time complexity of **insertLink**:  **$O(1)$** .

```
void insertLink(list my_list, link prev, link new_link){  
    if (prev == NULL) { // insert at beginning of list  
        new_link->next = my_list->first;  
        my_list->first = new_link;  
    }  
    else {  
        new_link->next = prev->next;  
        prev->next = new_link;  
    }  
    my_list->length += 1;  
}
```



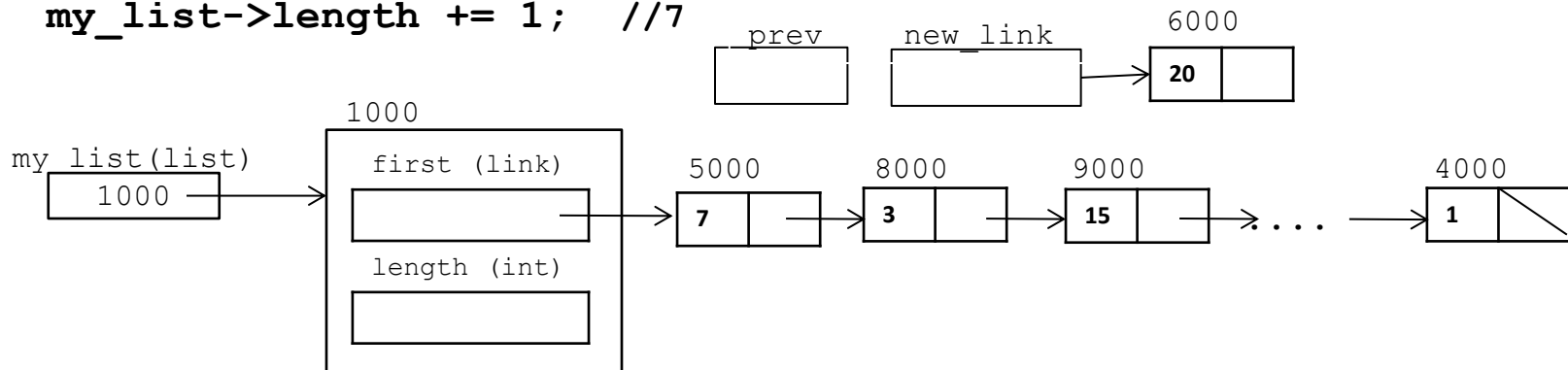
# Inserting a Link – Work Sheet

// Inserts the new link right after link **prev** (provided as argument).

// Time complexity of **insertLink**:  **$O(1)$** .

```
void insertLink(list my_list, link prev, link new_link){ //1
    if (prev == NULL) { // insert at beginning of list //2
        new_link->next = my_list->first; //3
        my_list->first = new_link; //4
    }
    else {
        new_link->next = prev->next; //5
        prev->next = new_link; //6
    }
    my_list->length += 1; //7
}
```

Show the actions for  
Inserting after  
prev = 8000  
Use the code line numbers  
as annotations to show  
what change they  
produced



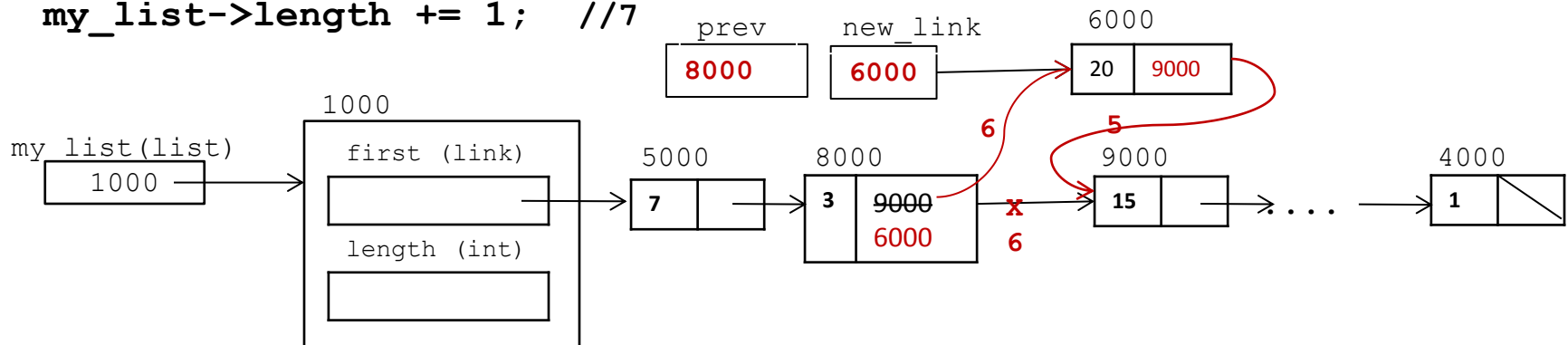
# Inserting a Link - Answer

// Inserts the new link right after link **prev** (provided as argument).

// Time complexity of **insertLink**:  $O(1)$ .

```
void insertLink(list my_list, link prev, link new_link){ //1
    if (prev == NULL) { // insert at beginning of list //2
        new_link->next = my_list->first; //3
        my_list->first = new_link; //4
    }
    else {
        new_link->next = prev->next; //5
        prev->next = new_link; //6
    }
    my_list->length += 1; //7
}
```

Show the actions for  
Inserting after  
prev = 8000  
Use the code line numbers  
as annotations to show  
what change they  
produced



# Inserting a Link

- Done:
  - Insert the new node right after **prev** (given as argument). -  $O(1)$
- Other useful functions for inserting a link?
  - Insert at a specific position, (instead of after a specific link).
  - Specify just the value in the new link, not the new link itself.
  - The above versions are individual practice.
  - What will their time complexity be?

# Removing a Link

```
void removeNext(list my_list, link x)
```

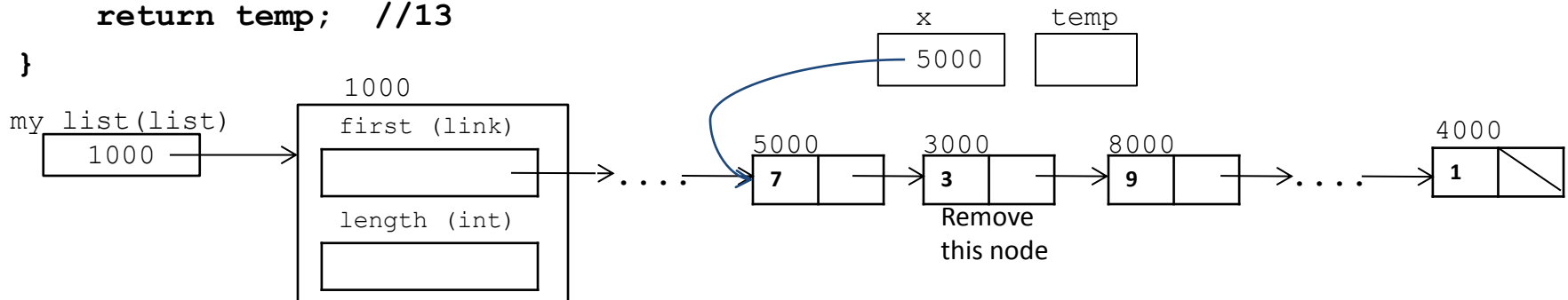
- The link **x** that we specify as an argument is NOT the link that we want to remove, but the link BEFOFE the one we want to remove.
  - If we know the previous link, we can easily access the link we need to remove.
  - The previous link needs to be updated to point to the next item.

# Removing a Link – Work sheet

```

link removeNext(list my_list, link x) { //1
    if (listIsNULL(my_list) == 1) { //2
        return NULL; //3
    }
    link temp; //4
    if (x == NULL) { // try to delete the first node //5
        temp = my_list->first; //6
        if (my_list->first != NULL) { // There is at least one node in the list. //7
            my_list->first = my_list->first->next; //8
            my_list->length -= 1; //9
        }
    } else {
        temp = x->next; //10
        x->next = x->next->next; // IS THIS CODE SAFE? JUSTIFY YOUR ANSWER. //11
        my_list->length -= 1; //12
    }
    return temp; //13
}

```

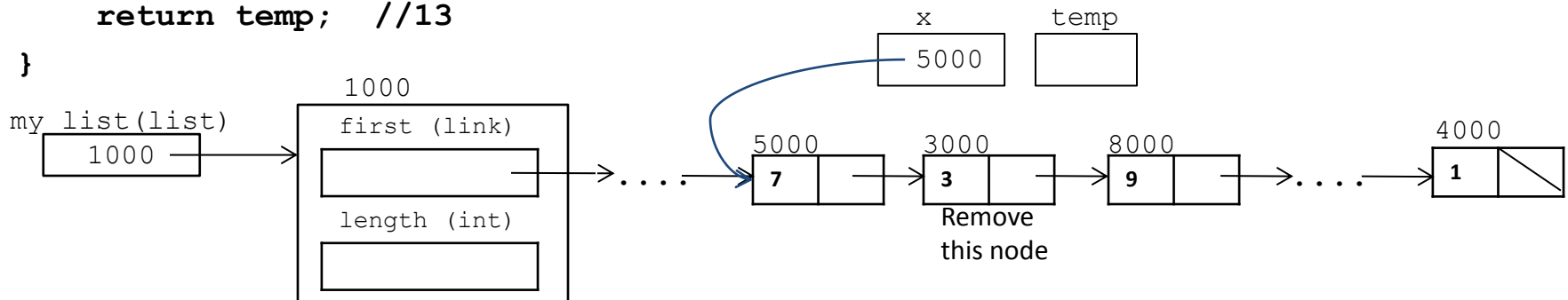


# Removing a Link

```

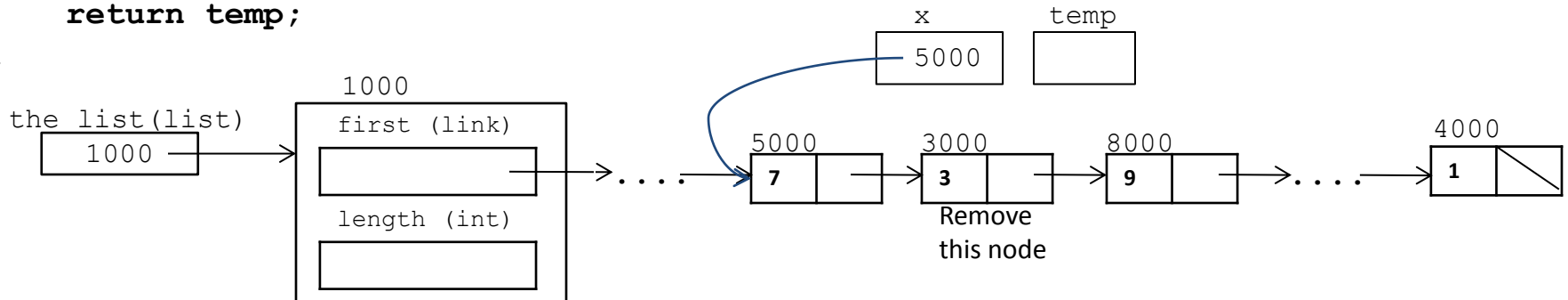
link removeNext(list my_list, link x) { //1
    if (listIsNULL(my_list) == 1) { //2
        return NULL; //3
    }
    link temp; //4
    if (x == NULL) { // try to delete the first node //5
        temp = my_list->first; //6
        if (my_list->first != NULL) { // There is at least one node in the list. //7
            my_list->first = my_list->first->next; //8
            my_list->length -= 1; //9
        }
    } else {
        temp = x->next; //10
        x->next = x->next->next; // IS THIS CODE SAFE? JUSTIFY YOUR ANSWER. //11
        my_list->length -= 1; //12
    }
    return temp; //13
}

```



# Removing a Link

```
link removeNext(list my_list, link x) {
    if (listIsNULL(my_list) == 1) {
        return NULL;
    }
    link temp;
    if (x == NULL) { // try to delete the first node
        temp = my_list->first;
        if (my_list->first != NULL) { // There is at least one node in the list.
            my_list->first = my_list->first->next;
            my_list->length -= 1;
        }
    } else {
        temp = x->next;
        x->next = x->next->next; //Still bad: using temp instead of x->next: x->next=temp->next)
        my_list->length -= 1;    // It crashes as soon as you 'execute' the second arrow.
    }
    // Crashes when x is the last node in the list: x->next is NULL.
    return temp;
}
```





# Removing a Link

- What is the time complexity of `removeLink`?  $O(1)$ .
  - Other useful version for removing a link
    - The argument, `x`, is the node that we want to delete. –
- Individual practice

# List – Work Sheet

- Write a function to swap the first two nodes in a list.
  - Write the code
  - Draw the picture
  - Identify the border cases or any other situation that could crash your code.
  - Fix the code so that it will not crash.

# More Pointer Issues....

- See the WrongPush and correct Push functions from the “Linked Lists Basics” (pages 12-15).
  - Note: the `head` in WrongPush is a new local variable, so if you draw it, make sure you draw another box (do not use the box for `head` from the test function).
- Does our list implementation (that uses a separate struct for the list) suffer from this problem?

# Reversing a List

```
void reverse(list the_list) //1
{
    link current = the_list->first; //2
    link previous = NULL; //3
    while (current != NULL) //4
    {
        link temp = current->next; //5
        current->next = previous; //6
        previous = current; //7
        current = temp; //8
    }
    the_list->first = previous; //9
}
```

# Example: Insertion Sort

- Here we do not modify the original list of numbers, we just create a new list for the result.

Algorithm: given *original* list, create *result* list

- For each number  $X$  in the *original* list:
  - Go through the *result* list (which is sorted), until we find the first node,  $A$ , with an item that is bigger than  $X$ .
  - Create a new node with value  $X$  and insert it right before node  $A$ .

# Insertion Sort Implementation

*/\* In this code, when is 'result->first' set to point to the actual first node in the new list? \*/*

```
list insertionSort(list numbers) {
    list result = newList();
    link s;
    for (s = numbers->first; s != NULL; s = s->next) {
        int value = s->item;
        link current = NULL;
        link next = result->first;
        while((next != NULL) && (value > next->item)) {
            current = next;
            next = next->next;
        }
        insertLink(result, current, newLink(value, NULL));
    }
    return result;
}
```

# listDeepCopy(...)

/\* In this code, when is 'result->first' set to point to the actual first node in the new list? \*/

```
list listDeepCopy(list input) {  
    list result = newList();  
    link in = getFirst(input);  
    link previous = NULL;  
    while (in != NULL) {  
        link out = newLink(getLinkItem(in), NULL);  
        insertLink(result, previous, out);  
        previous = out;  
        in = getLinkNext(in);  
    }  
    return result;  
}
```

# Summary: Lists vs. Arrays

Operation	Arrays	Lists
Access position $i$	$\Theta(1)$	$\Theta(i)$
Modify position $i$	$\Theta(1)$	$\Theta(i)$
Delete at position $i$	$\Theta(N-i)$ (worst case)	$\Theta(1)$ (assuming you are there)
Insert at position $i$	$\Theta(N-i)$ (worst case)	$\Theta(1)$ (assuming you are there)

- $N$ : length of array or list.
- The table shows time of worst cases.
- Other pros/cons:
  - When we create an array we must fix its size.
  - Lists can grow and shrink as needed.

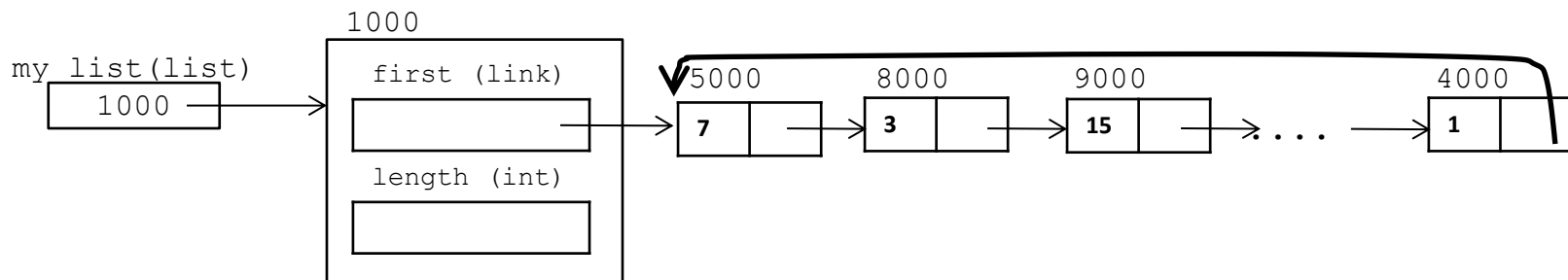


# Alternatives

- Circular lists
- Double-linked lists

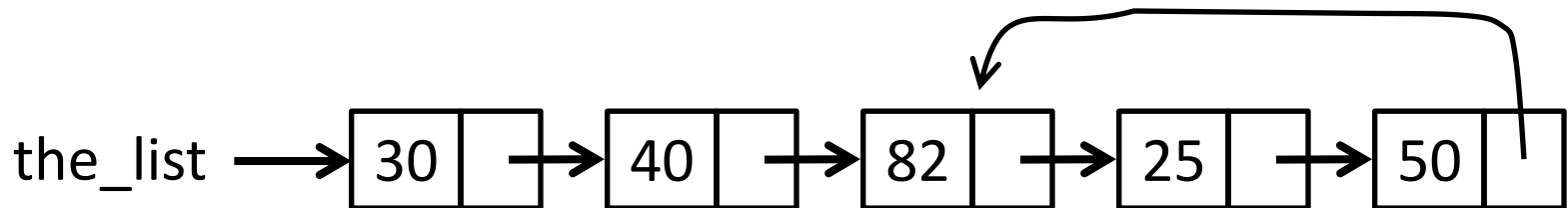
# Circular Lists

- The last node points back to the first node.
- Applications:
  - players in a board game,
  - round-robin assignments (e.g. take turns to allocate a resource to processes needing it), ...
  - Josephus-style election: arrange players in a circle, remove every M-th one, the last one wins.



# Detecting a Cycle in a Circular List

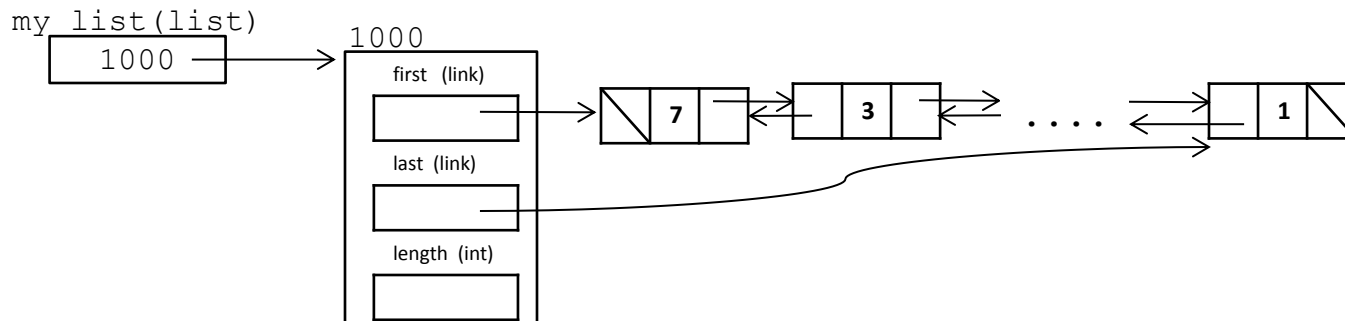
- Assume a list representation where you do NOT know the length of the list. E.g. the list is the pointer to the first node.
- Pb 1: Detect if a list has a cycle. No requirement of efficiency.
  - Have in mind that some initial items may not be part of the cycle:



- 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.

# Doubly-Linked Lists

- Each node *points both to the previous and to the next nodes*.
- In our list implementation, in addition to keeping track of the first node, we could also keep track of the last node.
- Advantages:
  - To delete a link, we just need that link.
  - Easy to go backwards and forward.
- Disadvantages:
  - More memory per link (one extra pointer).



# 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/>

# Food For Thought

- Assume you build a *sorted single (or double) linked list*
  - Every new element will be inserted in its right place.
  - There may be duplicate items in this list.
- You need to support as efficiently as possible:
  - `insert(my_list, val)` – create a new node with value `val` and insert it in the correct place in the list. If it is a duplicate (there is at least one other node with value `val`) it can be inserted before or after the node with the same value.
  - `delete(my_list, val)` – delete all nodes with value `val`.
  - `find(my_list, val)` - print all the nodes that have value `val` in them.
- **Can you do everyone of the above operations in  $O(\lg N)$**  by using binary search (and whatever else you need to be able to perform binary search)?
  - How would you do binary search, insert, delete?
  - Would it matter if you used a single or a double-linked list?