Elementary Data Structures: Part 1: Arrays, Lists

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
University of Texas at Arlington

Basic Types

- Types like integers, real numbers, characters. In C:
  - int
  - float
  - char
  - and variations: short, long, double, …
- Each basic type takes up a fixed amount of memory.
  - E.g: 32 bits for an int, 32 bits for a float, 8 bits for a char.
  - For C, this may vary, but the above values are common.
- Fixed memory implies limits in range, precision.
  - Integers above and below certain values are not allowed.
  - Real numbers cannot be specified with infinite precision.

Sets and Sequences

- A set is a very basic mathematical notion.
- Since this is not a math class, we can loosely say that a set is a collection of objects.
  - Some of these objects may be sets themselves.
- Sequences are **ordered sets**.
- In sequences, it makes sense to talk of:
  - first element, second element, last element.
  - previous element, next element.
- In sets, order does not matter.

Sets and Sequences in Programs

- It is hard to imagine large, non-trivial programs that do not involve sets or sequences.
- Examples where sets/sequences are involved:
  - Anything involving text:
    - Text is a sequence of characters.
  - Any database, that contains a set of records:
    - Customers.
    - Financial transactions.
    - Inventory.
    - Students.
    - Meteorological observations.
    - …
  - Any program involving putting items in order (sorting).

Representing Sets and Sequences

- Representing sets and sequences is a common and very important task in software design.
- Our next topic is to study the most popular choices for representing **sequences**.
  - Arrays.
  - Lists.
  - Strings.
- Arrays and lists can store arbitrary types of objects.
- Strings are custom-made to store characters.
- Each choice has its own trade-offs, that we need to understand.

Common Operations

- A data structure representing a sequence must support specific operations:
  - Initialize the sequence.
  - Delete the sequence.
  - Insert an item at some position.
  - Delete the item at some position.
  - Replace the item at some position.
  - Access (look up) the item at some position.
- The position (for insert, delete, replace, access) can be:
  - the beginning of the sequence,
  - or the end of the sequence,
  - or any other position.

Arrays

- In this course, it is assumed that you all are proficient at using arrays in C.
- IMPORTANT: the material in textbook chapter 3.2 is assumed to be known:
  - How to create an array.
  - How to access elements in an array.
  - Using **malloc** and **free** to allocate and de-allocate memory.
- Here, our focus is to understand the properties of array operations:
  - Time complexity.
  - Space complexity.
  - Other issues/limitations.

Array Initialization

- How is an array initialized in C?

Array Initialization

- How is an array initialized in C?
- If the size of the array is known when we write the code:

Array Initialization

- How is an array initialized in C?
- If the size of the array is known when we write the code:

```
int array_name[ARRAY_SIZE]
```

← static allocation

(where ARRAY_SIZE is a compile-time constant)

- If the size of the array is not known when we write the code:

Array Initialization

- How is an array initialized in C?
- If the size of the array is known when we write the code:

```
int array_name[ARRAY_SIZE]
```

static allocation

(where ARRAY_SIZE is a compile-time constant)

- If the size of the array is not known when we write the code:

```
int * array_name = malloc(ARRAY_SIZE * sizeof(int))
```

dynamic allocation

(where ARRAY_SIZE is a compile-time constant)

- Any issues/limitations with array initialization?

Array Initialization

- Major issue: the size of the array **MUST BE KNOWN** when the array is created.
- Is that always possible?

Array Initialization

- Major issue: the size of the array **MUST BE KNOWN** when the array is created.
- Is that always possible?
  - No, though it does happen some times.
- What do we do if the size is not known in advance?
  - What did the textbook do for the examples in Union-Find, Binary Search, and Selection Sort?

Array Initialization

- Major issue: the size of the array **MUST BE KNOWN** when the array is created.
- Is that always possible?
  - No, though it does happen some times.
- What do we do if the size is not known in advance?
  - What did the textbook do for the examples in Union-Find, Binary Search, and Selection Sort?
  - Allocate a size that (hopefully) is large enough.
- Problems with that:

Array Initialization

- Major issue: the size of the array **MUST BE KNOWN** when the array is created.
- Is that always possible?
  - No, though it does happen some times.
- What do we do if the size is not known in advance?
  - What did the textbook do for the examples in Union-Find, Binary Search, and Selection Sort?
  - Allocate a size that (hopefully) is large enough.
- Problems with allocating a "large enough" size:
  - Sometimes the size may not be large enough anyway.
  - Sometimes it can be a huge waste of memory.

Array Initialization and Deletion

- Time complexity of array initialization: constant time.

- How about array deletion? How is that done in C?
- If the array was statically allocated:
- If the array was dynamically allocated:
- Either way, the time complexity is: .

Array Initialization and Deletion

- Time complexity of array initialization: constant time.

- How about array deletion? How is that done in C?
- If the array was statically allocated: we do nothing.
- If the array was dynamically allocated: we call **free**.
- Either way, the time complexity is: *O(1)*.

Arrays: Inserting an Item

- "Inserting an item" for arrays can mean two different things.
- When the array is first created, it contains no items.
- The first meaning of "inserting an item" is simply to store a value at a position that previously contained no value.
- What is the time complexity of that?

Arrays: Inserting an Item

- "Inserting an item" for arrays can mean two different things.
- When the array is first created, it contains no items.
- The first meaning of "inserting an item" is simply to store a value at a position that previously contained no value.
- What is the time complexity of that? *O(1)*.

Arrays: Inserting an Item

- The second meaning of "inserting an item", which is the meaning we use in this course, is to insert a value at a position between other existing values.
- An example:
  – suppose we have an array of size 1,000,000.
  – suppose we have already stored value at the first 800,000 positions.
  – We want to store a new value at position 12,345, **WITHOUT** replacing the current value there, or any other value.
- We need to move a lot of values one position to the right, to make room.

Arrays: Inserting an Item

```
for (i = 800000; i >= 12345; i--)
   a[i] = a[i-1];
a[12345] = new_value;
```

- Why are we going backwards?

Arrays: Inserting an Item

```
for (i = 800000; i >= 12345; i--)
   a[i] = a[i-1];
a[12345] = new_value;
```

- Why are we going backwards?
  - To make sure we are not writing over values that we cannot recover.
- If the array size is N, what is the worst-case time complexity of this type of insertion?

Arrays: Inserting an Item

```
for (i = 800000; i >= 12345; i--)
   a[i] = a[i-1];
a[12345] = new_value;
```

- Why are we going backwards?
  - To make sure we are not writing over values that we cannot recover.
- If the array size is N, what is the worst-case time complexity of this type of insertion?
  - *O(N)*.

Arrays: Deleting an Item

- Again, we have an array of size 1,000,000.
  - We have already stored value at the first 800,000 positions.
  - We want to delete the value at position 12,345.
  - How do we do that?

Arrays: Deleting an Item

- Again, we have an array of size 1,000,000.
  - We have already stored value at the first 800,000 positions.
  - We want to delete the value at position 12,345.
  - How do we do that?

```
for (i = 12345; i < 800000; i++)
   a[i] = a[i+1];
```

- If the array size is N, what is the worst-case time complexity of deletion?

Arrays: Deleting an Item

- Again, we have an array of size 1,000,000.
  - We have already stored value at the first 800,000 positions.
  - We want to delete the value at position 12,345.
  - How do we do that?

```
for (i = 12345; i < 800000; i++)
   a[i] = a[i+1];
```

- If the array size is N, what is the worst-case time complexity of deletion?
  - *O(N)*.

Arrays: Replacing and Accessing

- How do we replace the value at position 12,345 with a new value?

```
a[12345] = new_value;
```

- How do we access the value at position 12,345?

```
int b = a[12345];
```

- Time complexity for both: *O(1)*.

Arrays: Summary

- Initialization: *O(1)* time, but must specify the size, which is a limitation.
- Deletion of the array: *O(1)* time, easy.
- Insertion: *O(N)* worst case time.
- Deletion of a single element: *O(N)* worst case time.
- Replacing a value: *O(1)* time.
- Looking up a value: *O(1)* time.
- Conclusions:
  - Arrays are great for looking up values and replacing values.
  - Initialization requires specifying a size, which is limiting.
  - Insertion and deletion are slow.

Linked Lists

- Many of you may have used lists, as they are built-in in many programming languages.
  – Java, Python, C++, …
- They are not built in C.
- Either way, this is the point in your computer science education where you learn to implement lists yourselves.

Contrast to Arrays

- An array is a contiguous chunk of memory.
  - That is what makes it easy, and fast, to access and replace values at specific positions.
  - That is also what causes the need to specify a size at initialization, which can be a problem.
  - That is also what causes insertion and deletion to be slow.
- Linked lists (as we will see in the next few slides) have mostly opposite properties:
  - No need to specify a size at initialization.
  - Insertion and deletion can be fast (though it depends on the information we provide to these functions).
  - Finding and replacing values at specific positions is slow.

The Notion of a Link

- When we create a list, we do not need to specify a size in advance.
  - No memory is initially allocated.
- When we insert an item, we allocate just enough memory to hold that item.
  - This allows lists to use memory very efficiently:
    - No wasting memory by allocating more than we need.
    - Lists can grow as large as they need (up to RAM size).
- Result: list items are not stored in contiguous memory.
  - So, how do we keep track of where each item is stored?
  - Answer: each item knows where the next item is stored.
  - In other words, each item is a **link** to the next item.

Links

```
typedef struct node * link;
struct node {Item item; link next;  };
```

- Note: the Item type can be defined using a **typedef**. It can be an int, float, char, or any other imaginable type.
- A **linked list** is a set of links.
  - This definition is simple, but **very important**.

Representing a List

- How do we represent a list in code?
- Initial choice: all we need is the first link. So, lists have the same type as links.
  - I don't like that choice, but we must first see how it works.
- How do we access the rest of the links?

Representing a List

- How do we represent a list in code?
- Initial choice: all we need is the first link. So, lists have the same type as links.
  - I don't like that choice, but we must first see how it works.
- How do we access the rest of the links?
  - Step by step, from one link to the next.
- How do we know we have reached the end of the list?

Representing a List

- How do we represent a list in code?
- Initial choice: all we need is the first link. So, lists have the same type as links.
  - I don't like that choice, but we must first see how it works.
- How do we access the rest of the links?
  - Step by step, from one link to the next.
- How do we know we have reached the end of the list?
  - Here we need a convention.
  - The convention we will follow: the last link points to NULL.

A First Program

```
#include <stdlib.h>
#include <stdio.h>

typedef struct node * link;
struct node  {int item; link next;  };

main()
{
   link the_list = malloc(sizeof(struct node));
   the_list->item = 573;
   the_list->next = NULL;
}
```

marking the end
of the list.

A First Program

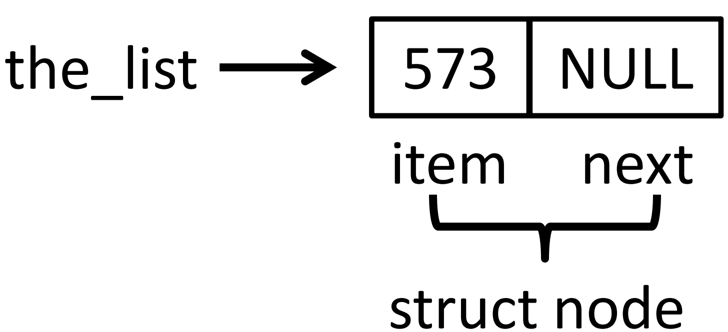
- What does the program in the previous slide do?
  - Not much. It just creates a list with a single item, with value 573.
- Still, this program illustrates some basic steps in creating a list:
  - There is no difference in the code between the list itself and the first link in the list.
  - To denote that there is only one link, the **next** variable of that link is set to **NULL**.
- Next: let's add a couple more links manually.

A Second Program

```
#include <stdlib.h>
#include <stdio.h>

typedef struct node * link;
struct node  {int item; link next;  };

link newLink(int value)
{  link result = malloc(sizeof(struct node));
   result->item = value;
   result->next = NULL;
}

main()
{  link the_list = newLink(573);
   the_list->next = newLink(100);
   the_list->next->next = newLink(200);
}
```

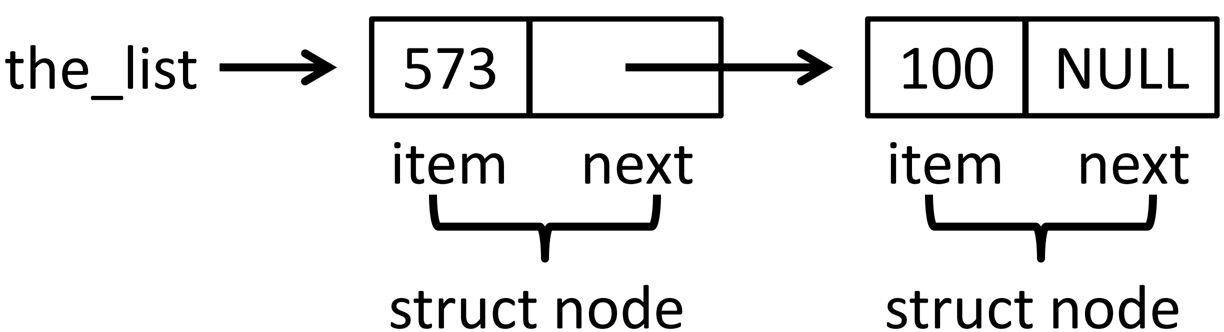
A Second Program

- What does the program in the previous slide do?
- It creates a list of three items: 573, 100, 200.
- We also now have a function **new_link** for creating a new link.
  - Important: by default, **new_link** sets the **next** variable of the result to NULL.
- How does the list look like when we add value 573?



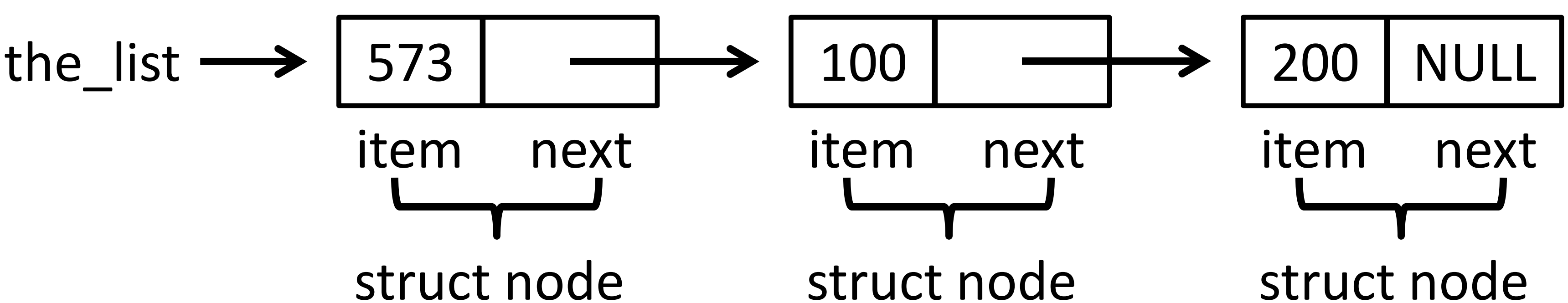
A Second Program

- What does the program in the previous slide do?
- It creates a list of three items: 573, 100, 200.
- We also now have a function **new_link** for creating a new link.
  – Important: by default, **new_link** sets the **next** variable of the result to NULL.
- How does the list look like when we add value 100?



A Second Program

- What does the program in the previous slide do?
- It creates a list of three items: 573, 100, 200.
- We also now have a function **new_link** for creating a new link.
  - Important: by default, **new_link** sets the **next** variable of the result to NULL.
- How does the list look like when we add value 200?



Printing the List

```
void print_list(link my_list)
{
    int counter = 0;
    link i;
    for (i = my_list; i != NULL; i = i->next)
    {
        printf("item %d: %d\n", counter, i->item);
        counter++;
    }
}
```

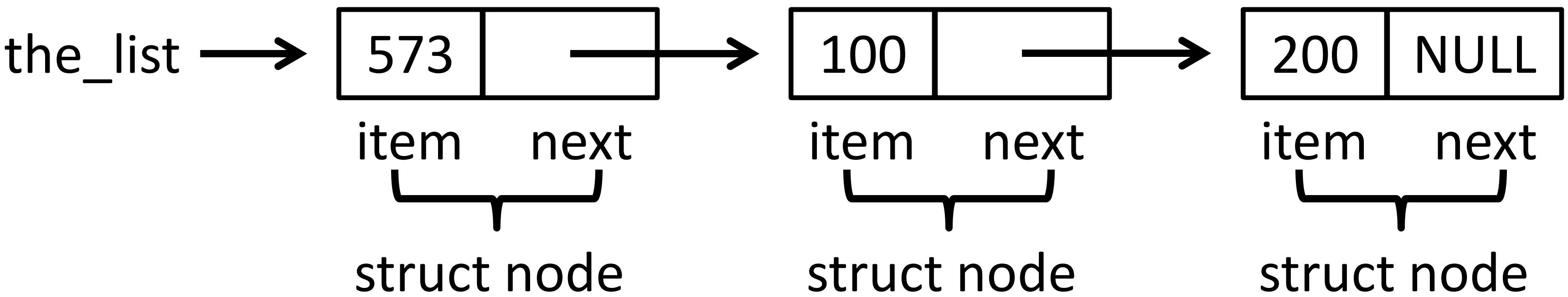
- The highlighted line in red is the CLASSIC way to go through all elements of the list. This is used EXTREMELY OFTEN.

Finding the Length of the List

```
int list_length(link my_list)
{
   int counter = 0;
   link i;
   for (i = my_list; i != NULL; i = i->next)
   {
      counter++;
   }
   return counter;
}
```

- The highlighted line in red is the CLASSIC way to go through all elements of the list. This is used EXTREMELY OFTEN.
- This kind of loop through the elements of a list is called **traversal of the list**.

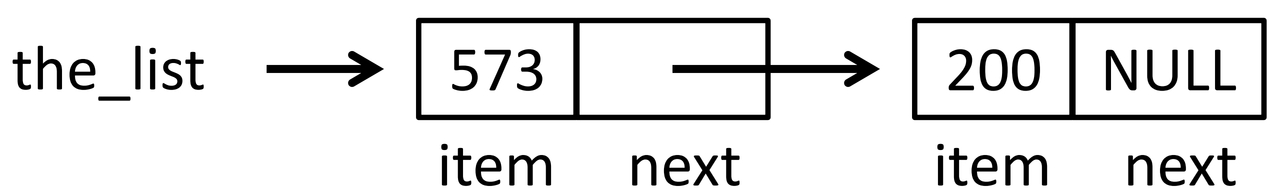
Deleting an Item



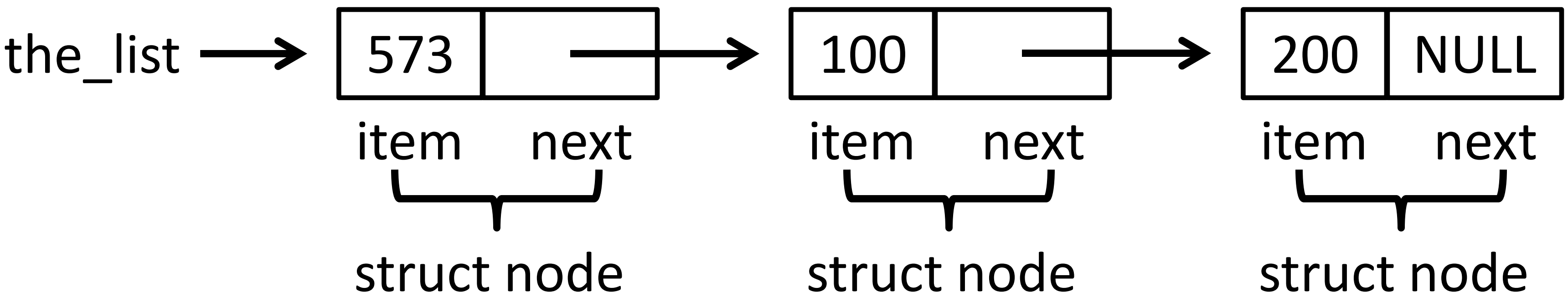
- Suppose that we want to delete the middle node. What do we need to do?
- Simple approach:

```
the_list->next = the_list->next->next;
```

Outcome:

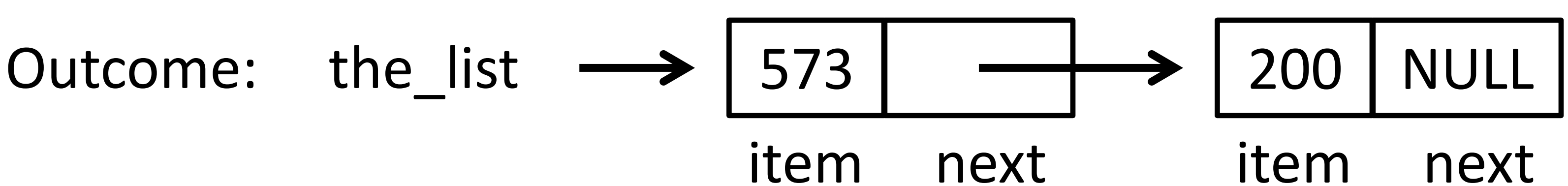


Deleting an Item

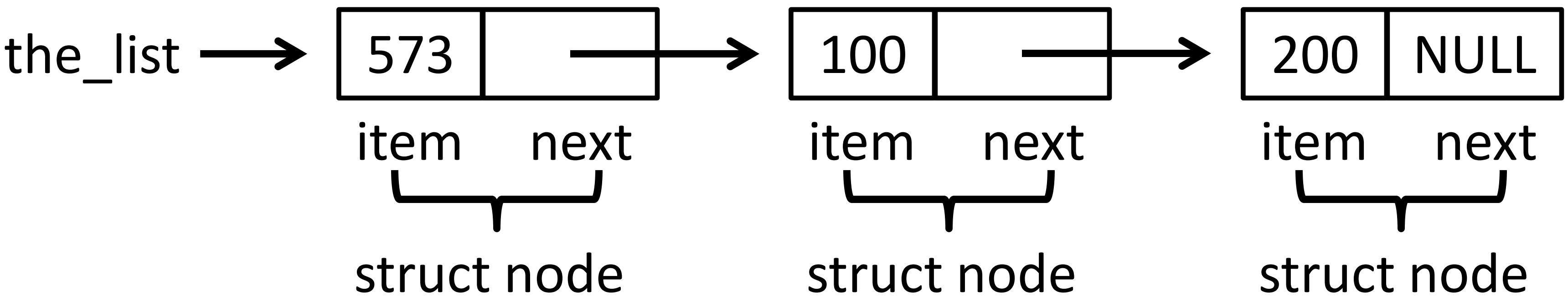


- Any problem with this approach?

```
the_list->next = the_list->next->next;
```



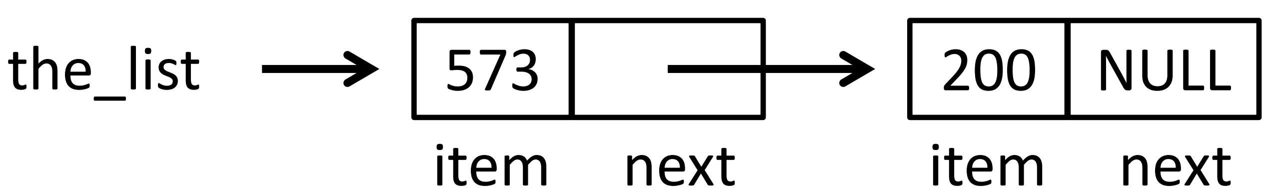
Deleting an Item



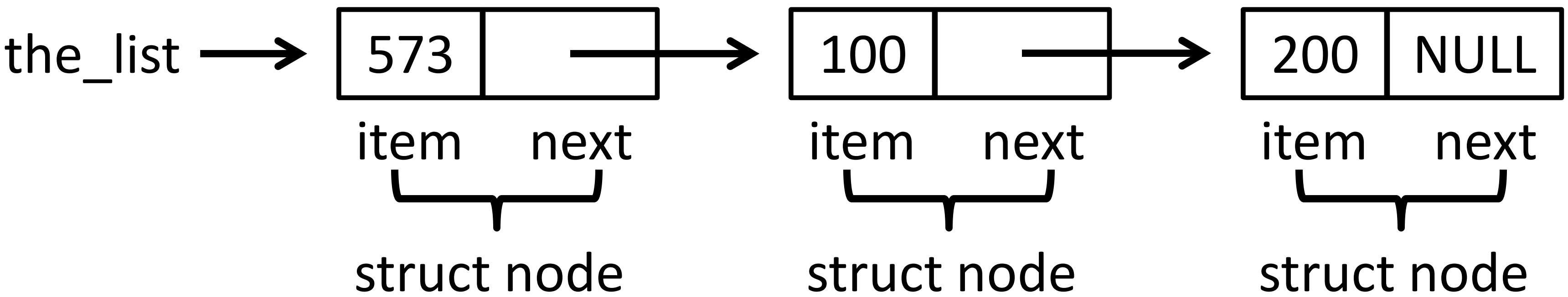
- Any problem with this approach? <u>**MEMORY LEAK**</u>

```
the_list->next = the_list->next->next;
```

Outcome:



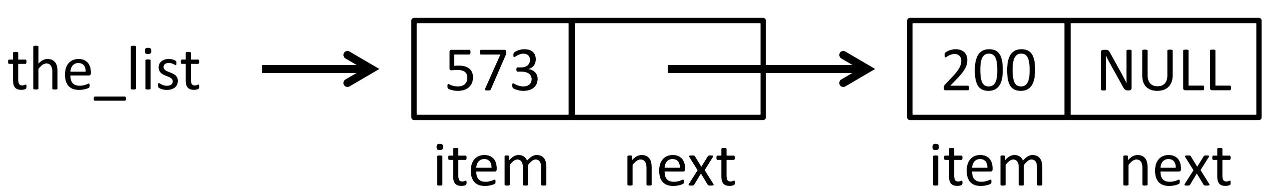
Deleting an Item



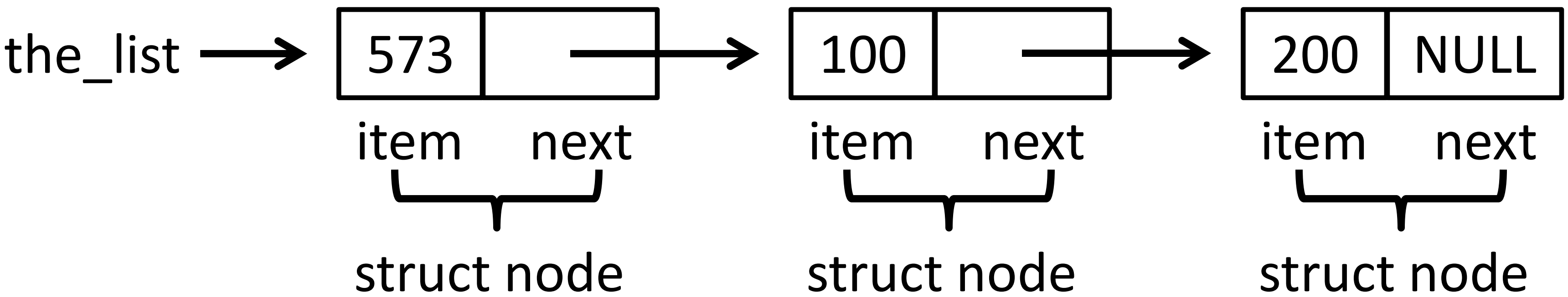
- Fixing the memory leak:

```
link temp = the_list->next;
the_list->next = the_list->next->next;
free(temp);
```

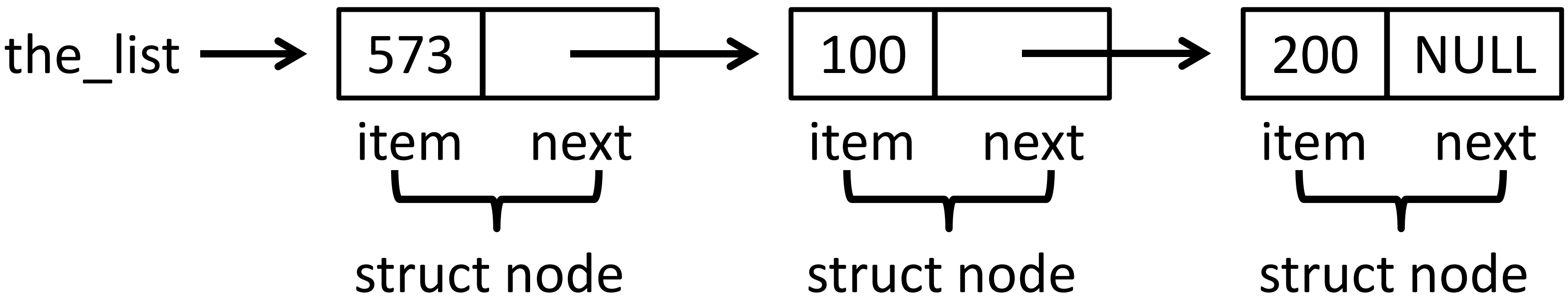
Outcome:



Deleting an Item from the Start

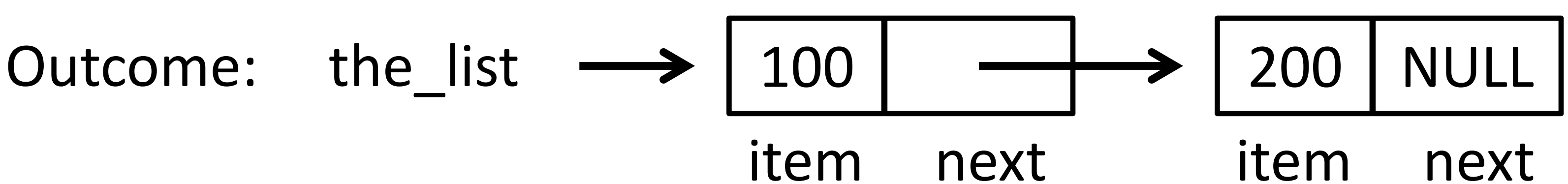


Deleting an Item from the Start

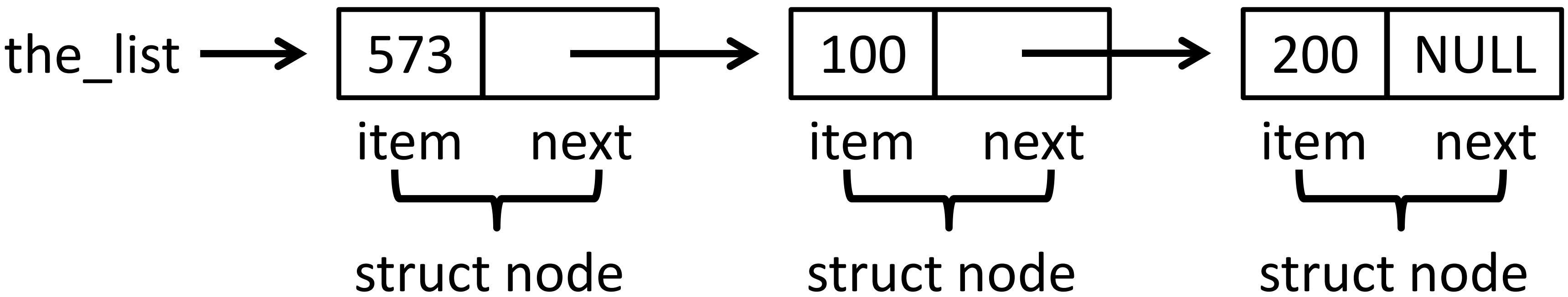


- This will work. Any issues?

```
link temp = the_list;
the_list = the_list->next;
free(temp);
```



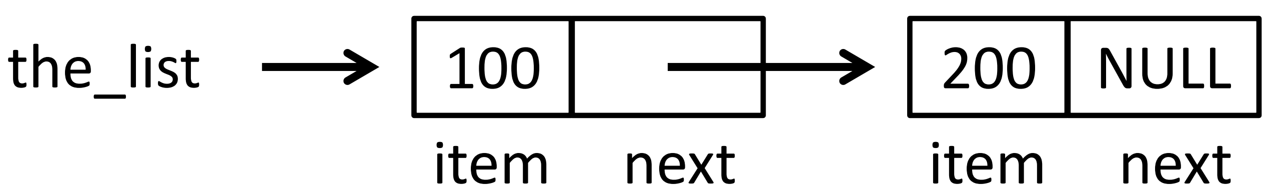
Deleting an Item from the Start



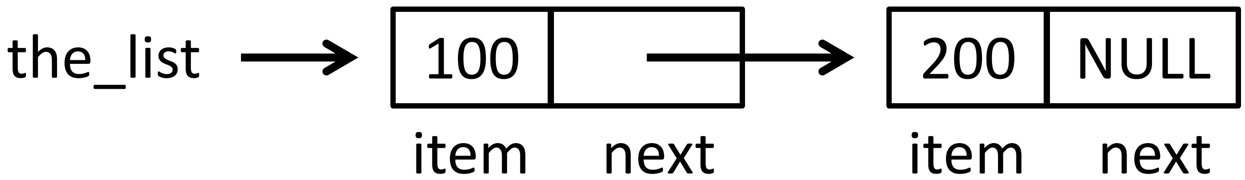
- This will work. Any issues? It is not that elegant.
  - We need to change the value of variable **`the_list`**.

```
link temp = the_list;
the_list = the_list->next;
free(temp);
```

Outcome:

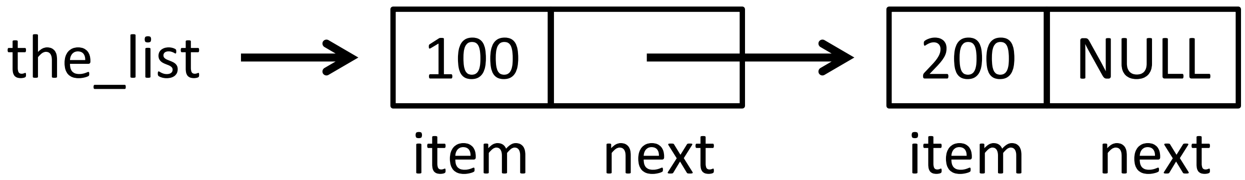


Inserting an Item



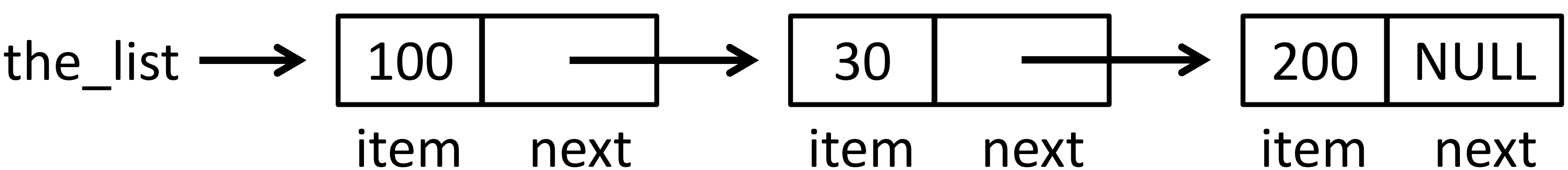
- Suppose we want to insert value 30 between 100 and 200. How do we do that?

Inserting an Item

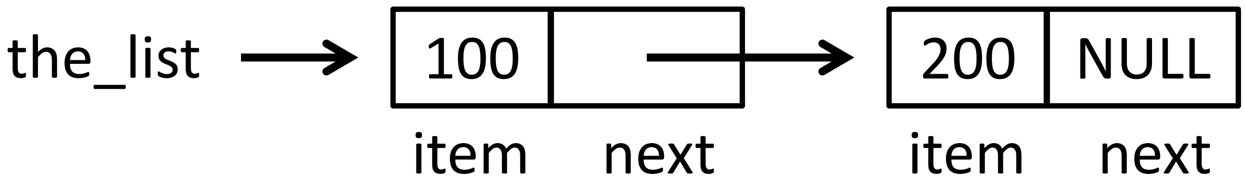


- Suppose we want to insert value 30 between 100 and 200. How do we do that?

```
link new_link = malloc(sizeof(struct node));
new_link->item = 30;
new_link->next = the_list->next;
the_list->next = new_link;
```

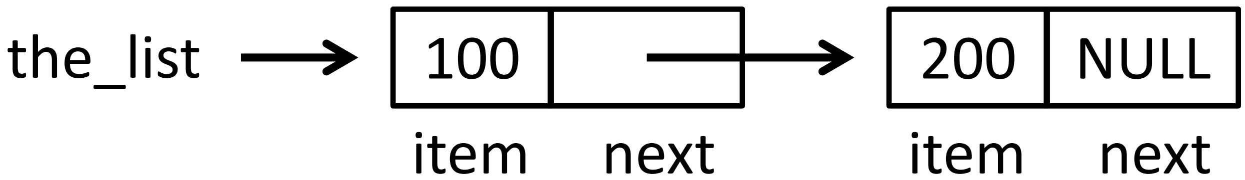


Inserting an Item to the Start



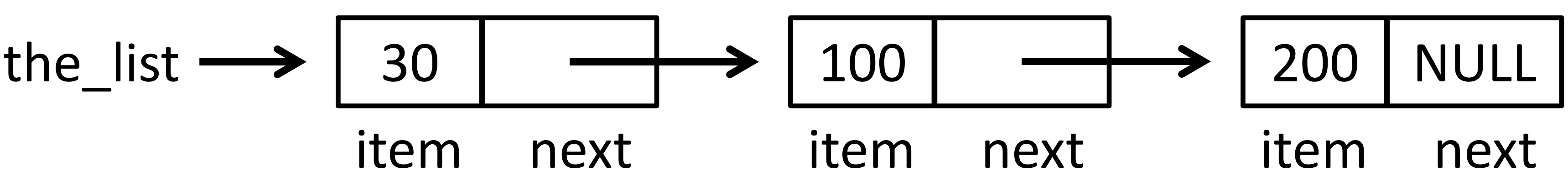
- Suppose we want to insert value 30 at the start of the list:

Inserting an Item to the Start

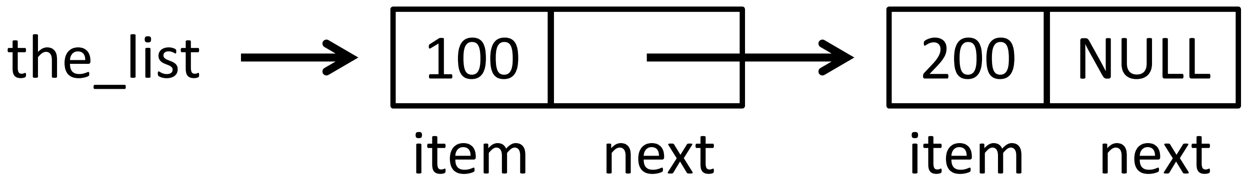


- Suppose we want to insert value 30 at the start of the list:

```
link new_link = malloc(sizeof(struct node));
new_link->item = 30;
new_link->next = the_list;
the_list = new_link;
```

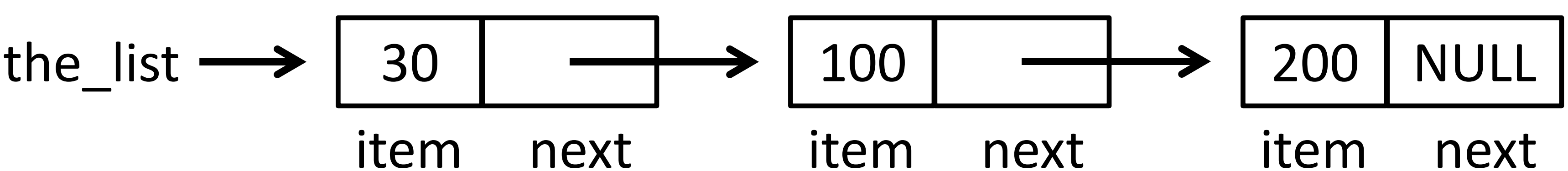


Inserting an Item to the Start



- Suppose we want to insert value 30 at the start of the list:
- Any issues with this code? Again, it is inelegant.
  - As in deleting from the start, we need to change variable **`the_list`**.

```
link new_link = malloc(sizeof(struct node));
new_link->item = 30;
new_link->next = the_list;
the_list = new_link;
```



An Example: Reading Integers

```
#include <stdlib.h>
#include <stdio.h>

typedef struct node * link;
struct node  {int item; link next;  };

main()
{  link the_list = NULL, current_link = NULL;
   while(1)
   {  int number;
      printf("please enter an integer: ");
      if (scanf("%d", &number) != 1) break;
      link next_item = malloc(sizeof(struct node));
      next_item->item = number; next_item->next = NULL;
      if (the_list == NULL) the_list = next_item;
      else current_link->next = next_item;
      current_link = next_item;
   }
}
```

Lists: What We Have Done So Far

- Defined a linked list as a set of links.
- Each link contains enough room to store a value, and to also store the address of the next link.
  - Why does each link need to point to the next link? Because otherwise we would not have any way to find the next link.
- Convention: the last link points to NULL.
- Insertions and deletions are handled by updating the link before the point of insertion or deletion.
- The variable for the list itself is set equal to the first link.
  - This is workable, but hacky and leads to inelegant code.

Lists: Next Steps

- Change our convention for representing the list itself.
  - Decouple the list itself from the first link of the list.
- Provide a set of functions performing standard list operations.
  - Initialize a list.
  - Destroy a list.
  - Insert a link.
  - Delete a link.

Representing a List

- First choice: a list is equal to the first link of the list.
- This is hacky. Conceptually, a variable representing a list should not have to change because we insert or delete a link at the beginning.
- The book proposes the "dummy link" solution, which I also don't like as much:
  - The first link of a list is always a dummy link, and thus it never has to change.
- The code in the book uses this solution.
- In class we will use another solution: lists and links are different data types.

The New List Representation

```
typedef struct struct_list * list;
struct struct_list
{  link first;  };

list newList(): ???
```

The New List Representation

```
typedef struct struct_list * list;
struct struct_list
{  link first;  };

list newList()
{
   list result = malloc(sizeof(*result));
   result->first = NULL;
   return result;
}
```

Destroying a List

- How do we destroy a list?

```
void destroyList(list 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;
      free(i);
      i = next;
   }
   free(the_list);
}
```

Inserting a Link

- How do 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

```
void insertLink(list my_list, link prev, link new_link)
{
    if (prev == NULL)
    {
        new_link->next = my_list->first;
        my_list->first = new_link;
    }
    else
    {
        new_link->next = prev->next;
        prev->next = new_link;
    }
}
```

Inserting a Link

- What is the time complexity of **`insertLink`**?

Inserting a Link

- What is the time complexity of **`insertLink`**? *O(1)*.

Inserting 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.
- What other functions for inserting a link may be useful?

Inserting 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.
- What other functions for inserting a link may be useful?
  - Specifying the position, instead of the previous link.
  - Specifying just a value for the new link, instead of the new link itself.

Deleting a Link

- How do we delete a link?

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

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

Deleting a Link

```
void deleteNext(list my_list, link x)
{
    link temp = x->next;
    x->next = temp->next;
    free(temp);
}
```

Deleting a Link

- What is the time complexity of **`deleteLink`**?
- What are the limitations of this version of deleting a link?
- What other versions of deleting a link would be useful?

Deleting a Link

- What is the time complexity of **`deleteLink`**? *O(1)*.
- What are the limitations of this version of deleting a link?
  - We cannot delete the first link of the list.
- What other versions of deleting a link would be useful?
  - Passing as an argument the node itself that we want to delete.
  - How can that be implemented?

Reversing a List

```
void reverse(list the_list)
{
   link current = the_list->first;
   link previous = NULL;
   while (current != NULL)
   {
      link temp = current->next;
      current->next = previous;
      previous = current;
      current = temp;
   }
   the_list->first = previous;
}
```

Example: Insertion Sort

- Unlike our implementation for Selection Sort, here we do not modify the original list of numbers, we just creates a new list for the result.
- For each number *X* in the original list:
  - Go through the result list, until we find the first item *Y* that is bigger than *M*.
  - Insert *X* right before that item *Y*.

Insertion Sort Implementation

```
list insertionSort(list numbers)
{
   list result = newList();
   link s;
   for (s = numbers->first; s!= NULL; s = s->next)
   {
      int value = s->item;
      link current = 0;
      link next = result->first;
      while((next != NULL) && (value > next->item))
      {
         current = next;
         next = next->next;
      }
      insertLink(result, current, newLink(value));
   }
   return result;
}
```

Doubly-Linked Lists

- In our implementation, every link points to the next one.
- We could also have every link point to the previous one.
- Lists where each link points both to the previous and to the next element are called **doubly-linked lists**.
- The list itself, in addition to keeping track of the first element, could also keep track of the last element.
- Advantages:
  - To delete a link, we just need that link.
  - It is as easy to go backwards as it is to go forward.
- Disadvantages:
  - More memory per link (one extra pointer).

Summary: Lists vs. Arrays

Operation	Arrays	Lists
Access position *i*	*O(1)*	*O(i)*
Modify position *i*	*O(1)*	*O(i)*
Delete at position *i*	*O(N)*	*O(1)*
Insert at position i	*O(N)*	*O(1)*

- 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.

Abstracting the Interface

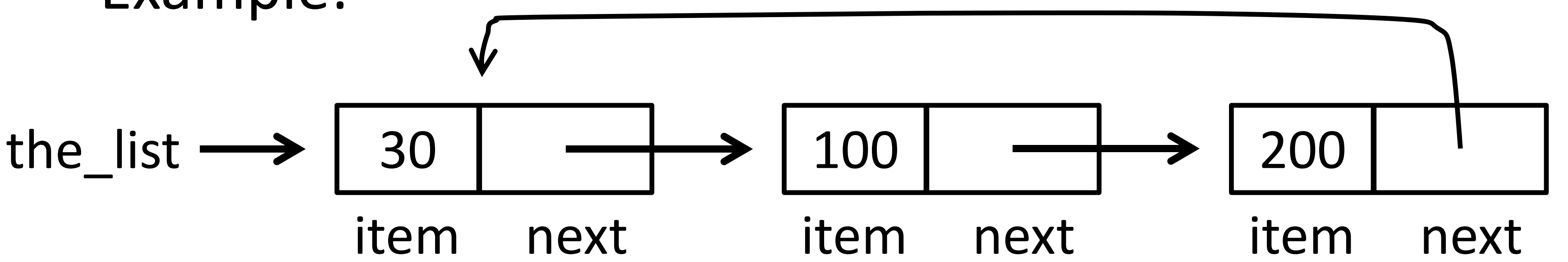
- When designing a new data type, it is important to hide the details of the implementation from the programmers who will use this data type (including ourselves).
- Why? So that, if we later decide to change the implementation of the data type, no other code needs to change besides the implementation.
- In C, this is doable, but somewhat clumsy.
- C++ and Java were designed to make this task easy.
  - By allowing for member functions.
  - By differentiating between private and public members.

List Interface

- The following files on the course website implement an abstract list interface:
  - list_interface.h
  - list_interface.c
- Other code that wants to use lists can only see what is declared at list_interface.h.
  - The actual implementation of lists and nodes is hidden.
- The implementation in list_interface.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.

Circular Lists

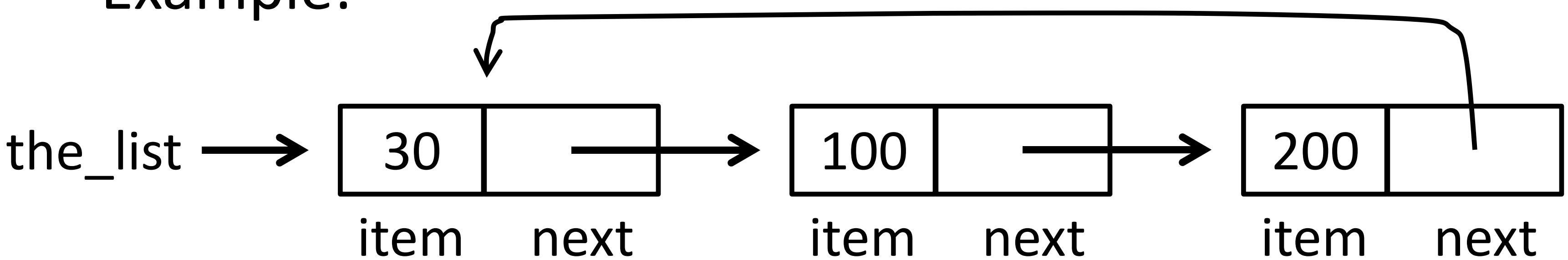
- What is a circular list? It is a list where some link points to a previous link.
- Example:



- When would a circular list be useful?

Circular Lists

- What is a circular list? It is a list where some link points to a previous link.
- Example:



- When would a circular list be useful?
  - In representing items that can naturally be arranged in a circular order.
  - Examples: months of the year, days of the week, seasons, players in a board game, round-robin assignments, …

The Josephus-Style Election

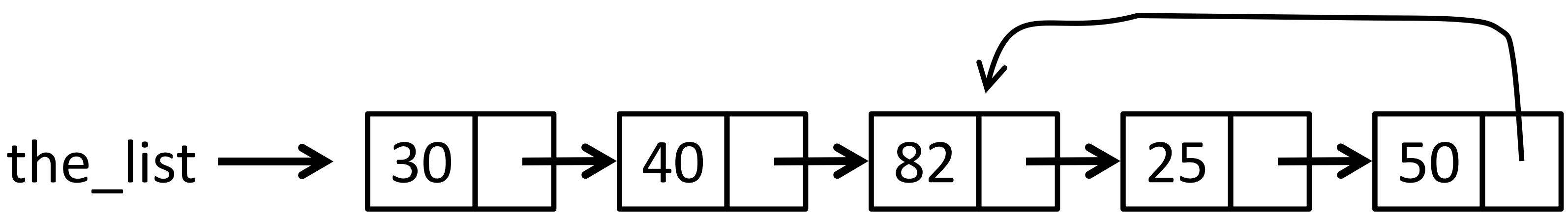
- This is a toy example of using circular lists.
- *N* people want to elect a leader.
  - They choose a number *M*.
  - They arrange themselves in a circular manner.
  - Starting from some person, they count *M* people, and they eliminate the *M*-th person. That person falls out of the circle.
  - Start counting again, starting from the person right after the one who got eliminated, and eliminate the *M*-th person again.
  - Repeat till one person is left.
- The last person left is chosen as the leader.

Implementing Josephus-Style Election

- If we assign numbers 1 to N to the N people, and we start counting from person 1, then the result is a function of N and M.
- This process of going around in a circle and eliminating every M-th item can be handled very naturally using a circular list.
- Solution: see josephus.c file, posted on course website.
- Note: our abstract interface was built for NULL-terminated lists, not circular lists.
- Still, with one change and one hack (marked on the code), it supports circular lists, at least for the purposes of the Josephus problem.
  - Change: in deleteNext, handle the case where we delete the first link.
  - Hack: make the list NULL-terminated before we destroy it.

Circular Lists: Interesting Problems

- There are several interesting problems with circular lists:
  - Detect if a list is circular.
    - Have in mind that some initial items may not be part of the cycle:



  - Detect if a list is circular **<u>in *O(N)* time</u>** (*N* is the number of unique nodes). (This is a good interview question)
  - Modifying our abstract list interface to fully support circular lists.
    - Currently, at least these functions would not support it: listLength, printList, destroyList, reverse.

Destructive Functions

```
void insertLink(list my_list, link prev, link new_link);
void deleteNext(list my_list, link x);
void reverse(list the_list);
```

- We call a function **destructive** if it modifies one or more of its input arguments.
- Several of the list functions we have seen are destructive (see examples above).
- We use destructive functions frequently, because they have attractive properties in terms of time and space requirements.
- However, when using destructive functions we must be aware of certain issues.

Issues with Destructive Functions

- The input argument that is modified may be accessible by many parts of the code.
- A common source of bugs is to modify such an input argument, and then assume (in another part of the code) that it has not been modified.
- Using a destructive function requires the programmer to be aware of all possible ramifications.
  - This can be very complicated, in a large program.
- On the other hand, using non-destructive functions makes our life much more simple.
- Then, why do we use destructive functions?
  - Because some times they are far more efficient than other alternatives.

Shallow and Deep Copies

- We say that B is (at some particular moment) a shallow copy of A if:
  - B, at that moment, contains the same information as A.
  - It is possible for changes in A to change B as well, or …
  - it is possible for changes in B to change A as well.
- We say that B is (at some particular moment) a deep copy of A if:
  - B, at that moment, contains the same information as A.
  - If A changes later, B is not affected.
  - If B changes later, A is not affected.

Example: List Deep Copy

```
list listDeepCopy(list input)
{
   list result = newList();
   link in = listFirst(input);
   link previous = NULL;
   while (in != NULL)
   {
      link out = newLink(linkItem(in));
      insertLink(result, previous, out);
      previous = out;
      in = linkNext(in);
   }
   return result;
}
```

Writing Non-Destructive Functions

- If we want to convert a destructive function to a non-destructive function, a common strategy is:
  - Identify the input arguments that the destructive function changes.
  - In the non-destructive version, make deep copies of those input arguments, and make changes to those deep copies.
  - Possibly return some of those modified deep copies, so they can be used by callers of the function.

Example: Destructive mergeLists

- Write a function that:
  - takes two arguments, a list **target**, and a list **source**.
  - adds all contents of **source** to **target**, effectively merging **source** to **target**.
- Note: at the end of the function, **target** has been changed, to be the result of merging the initial contents of **target** with the contents of **source**.

```
void mergeListsDestructive(list target, list source)
```

Example: Destructive mergeLists

```
void mergeListsDestructive(list target, list source)
{
   link previous = NULL;
   link c;

   /* find the last link of target*/
   for (c = target->first; c != NULL; c = c->next)
   {
      previous = c;
   }

   /* now, previous is the last link of target */
   setNext(previous, listFirst(source));
}
```

Example: Non-Destructive mergeLists

- Write a function that:
  - takes two arguments, a list **input1**, and a list **input2**.
  - returns a new list, that contains all contents of **input1** and all contents of **input2**.
  - does not change the input arguments.

```
list mergeLists(list input1, list input2)
```

Example: Non-Destructive mergeLists

```
list mergeLists(list input1, list input2)
{
   list result = listDeepCopy(input1);
   list temp2 = listDeepCopy(input2);
   mergeListsDestructive(result, temp2);
   free(temp2);
   return result;
}
```

Non-Destructive Insertions?

```
void insertLink(list list1, link prev, link link1)
```

- How can we make a non-destructive version of insertLink?

- What would be the time complexity of the non-destructive version?

- Why is the destructive version more popular?

Non-Destructive Insertions?

```
void insertLink(list list1, link prev, link link1)
```

- How can we make a non-destructive version of insertLink?
  - Make a deep copy of the list
  - Insert the new link to the deep copy.
  - Return the deep copy (that now includes the new link).
- What would be the time complexity of the non-destructive version?
  - *O(N)*, where *N* is the length of the list.
- Why is the destructive version more popular?
  - It takes *O(1)* time, and also does not duplicate memory.