

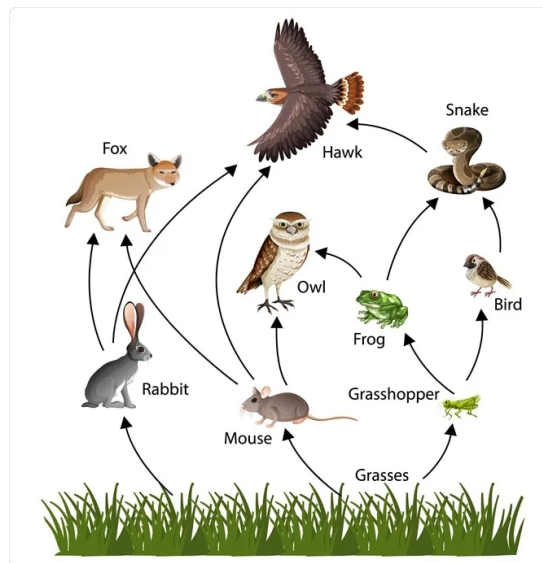
Instructor created

14.2 Project 02 - Food Web Analysis with Dynamic Memory

Introduction

In this project, food web analysis is performed by taking user-specified predator-prey relationships to build a dynamically allocated struct array of organisms, which is then used to identify the relationships between the organisms in the food web. Then, the analysis is redone after one of the species goes extinct, which requires completely removing that organism from the food web. That is, the array of organisms AND the set of predator-prey relationships must both be rearranged and reallocated to remove all signs of the extinct species.

Figure 14.2.1: Sample Food Web



Organism Struct & Dynamic Arrays

For the purposes of this project, an organism is characterized by its struct subitems, as defined in the starter code as follows:

```
typedef struct Org_struct {
    char name[20];
    int* prey; //dynamic array of indices
    int numPrey;
} Org;
```

The **Org** struct contains the **name** of the organism as a character array (i.e. a *string*) and a dynamic int array representing its **prey** (i.e. *what it eats*) in the food web. The size of the **prey** array is stored as the **numPrey** subitem.

The organisms of the food web are stored in a dynamic array of **Org** structs, called **web**, which has size **numOrgs** that is read-in from user-input. The starter code includes the allocation and initialization of the **web** array in **main()**. Note that all organisms are initialized with a name read-in from user-input and no prey (i.e. an empty array of zero size is simply a pointer to **NULL**):

```
printf("Enter names for %d organisms:\n", numOrgs);
for (int i = 0; i < numOrgs; ++i) {
    scanf("%s", web[i].name);
    web[i].prey = NULL;
    web[i].numPrey = 0;
}
```

For the remainder of the program, the individual organisms are uniquely identified by their **index** in the **web** array, which maintains the order in which the organisms were read-in from user-input.

The predator-prey relationships among the organisms are then read-in from user-input in the format

[predator index] [prey index]

where the indices refer to the **web** array. For each predator-prey relationship, the **prey index** should be appended to the predator's **prey** array. That is, the organism at **predator index** in **web** must have its **prey[]** array reallocated to allow an additional entry, specifically **prey index**.

Example Food Web

As an example, consider the Grassland Food Web, which contains the following 7 organisms (indices precede the organism name):

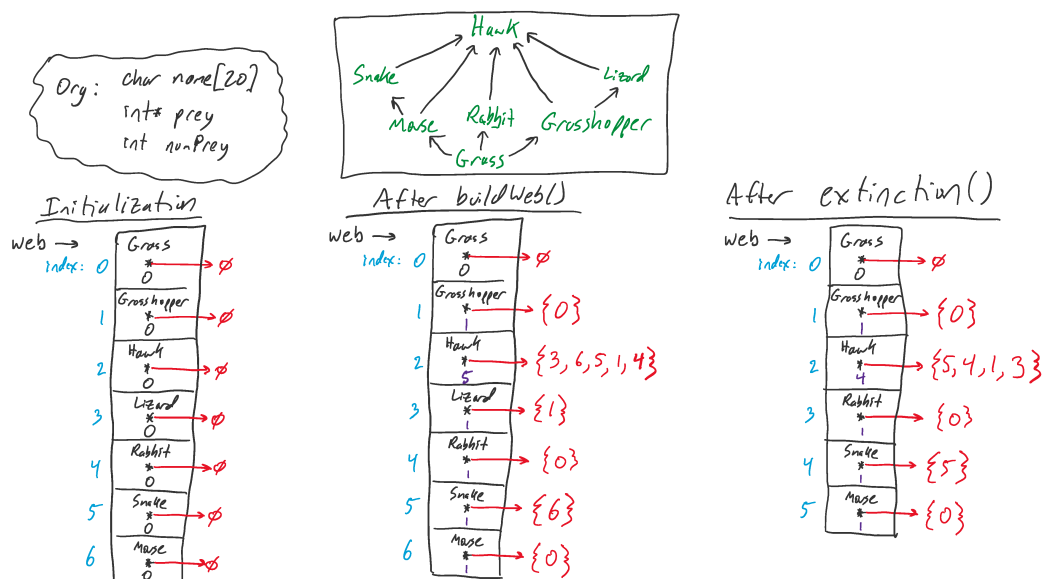
- 0: Grass
- 1: Grasshopper
- 2: Hawk
- 3: Lizard
- 4: Rabbit
- 5: Snake
- 6: Mouse

Now, consider the following 10 predator-prey relationships (with index relationship shown in parentheses):

- Mouse eats Grass (6 <- 0)
- Lizard eats Grasshopper (3 <- 1)
- Hawk eats Lizard (2 <- 3)
- Hawk eats Mouse (2 <- 6)
- Hawk eats Snake (2 <- 5)
- Rabbit eats Grass (4 <- 0)
- Hawk eats Grasshopper (2 <- 1)
- Snake eats Mouse (5 <- 6)
- Hawk eats Rabbit (2 <- 4)
- Grasshopper eats Grass (1 <- 0)

The figure below shows how the organisms are stored in a dynamically allocated **Org** struct array called **web**, where each **Org** has a dynamically allocated int array called **prey**, which is the list of indices in **web** for which that organism eats. More on extinction later...

Figure 14.2.2: Sample Woodland Food Web and its Dynamic Org Arrays with Dynamic prey Arrays



Starter Code & Input Files

The starter code contains the **Org** struct definition, function headers for two required functions (**buildWeb** and **extinction**), the initial setup of the **web[]** array built from user-input, and a scaffolding of all required steps in **main()**. Look for the TODO statements in the starter code to align them with the tasks below. All required `scanf()` statements are provided in the starter code, to ensure all user-input is read into the program in the proper order and format. This allows us to use set predefined input files that contain all of the user input for a given food web. Thus, compile the code and use a file to input values to the `scanf()` calls directly using a redirection operator (`<`) as follows:

```
gcc main.c -o prog.out
./prog.out < GrasslandFoodWeb.txt
```

Programming Task I - buildWeb()

Your first task is to write the **buildWeb()** function, which adds a predator-prey relation to the food web. In the starter code, `buildWeb()` is called from `main()` for each [predator index] [prey index] pair read in from user-input, representing a single predator-prey relationship. This function requires three parts:

1. **Memory allocation:** if the predator's **prey[]** array is empty, allocate memory for one index; otherwise, reallocate the predator's **prey[]** array to allow one more additional index. Do NOT use `realloc()`; instead, the reallocation should be performed manually by `malloc`'ing a new array, copying the elements to the new array, freeing up the old array, and reassigning the array pointer to the new array.
2. **Append the new prey:** append the prey index as the last element of the predator's **prey[]** array
3. **Update size of prey[] array:** update the **numPrey** subitem for the predator appropriately

Make sure the memory allocated for each `prey[]` array is ALWAYS the exact (i.e. no extra wasted memory, but not too little) amount of memory needed to store the prey indices for each organism; and that `numPrey` ALWAYS accurately represents the size of the `prey[]` array.

To check if the web is built correctly, write additional function(s) to print the **web** in the format consistent with this sample output (for `GrasslandFoodWeb.txt`):

```
Food Web Predators & Prey:
Grass
Grasshopper eats Grass
Hawk eats Lizard, Mouse, Snake, Grasshopper, Rabbit
Lizard eats Grasshopper
Rabbit eats Grass
Snake eats Mouse
Mouse eats Grass
```

Note that all organisms are printed in the order they were read-in from user-input and stored in the `web[]` array. Organisms that eat no other organisms are simply printed, whereas organisms with at least one prey are printed with a formatted list of what they eat, again in the order that the predator-prey relation was entered by the user.

Programming Task II - Food Web Analysis

Now that the food web is built and printed correctly, it is time to perform the following analyses:

- identify apex predators,
- identify producers
- identify the most flexible eaters
- identify the tastiest food
- calculate the height of each organism in the food web
- categorize each organism as a producer, herbivore, omnivore, or carnivore

Each analysis is described in the following subsections. Each step should be decomposed into separate function(s) and called from `main()` in the appropriate place in the starter code. This entire task will involve properly accessing elements of the `web[]` array and the `prey[]` array subitems.

REMINDER: whereas dynamically allocated arrays are declared using a pointer, e.g.

```
Org* web = (Org*)malloc(numOrgs*sizeof(Org));
```

the square bracket access/assignment operator can be used just like a traditional array, e.g.

```
scanf("%s",web[i].name); //assigns name for ith Org in web using user-input
```

Note: all printed outputs of multiple organisms should be in the order they were first entered by user, which is in alphabetical order for provided input .txt files. The predator-prey relationships should also be printed in the order they were read in from the user, which has a random order in the provided input .txt files.

Identify Apex Predators

An apex predator is an organism that is not the prey of any other organism in the food web. For the Grassland Food Web example, the output is as follows:

Apex Predators:

Hawk

There may be more than one apex predator, in which case print them in the order they were first entered by the user. For the Aquatic Food Web, the output is as follows:

Apex Predators:

Bird

Fish

Lobster

Identify Producers

A producer is an organism that does not eat any other organism in the food web (typically, producers are plants and get their energy directly from the sun). For the Grassland Food Web example, the output is as follows:

Producers:

Grass

There may be more than one producer, in which case print them in the order they were first entered by the user. For the Aquatic Food Web, the output is as follows:

Producers:

Phytoplankton

Seaweed

Identify the Most Flexible Eaters

The most flexible eater is the organism(s) that eat the greatest number of different organisms in the food web. For the Grassland Food Web example, the output is as follows:

Most Flexible Eaters:

Hawk

There may be a tie for the most flexible eater, in which case print them in the order they were first entered by the user. For the Mixed Food Web, the output is as follows:

Most Flexible Eaters:

Hawk

Owl

Identify the Tastiest Food

The tastiest food is the organism(s) that get eaten by the greatest number of different organisms in the food web. For the Grassland Food Web example, the output is as follows:

Tastiest Food:

Grass

There may be a tie for the tastiest food, in which case print them in the order they were first entered by the user. For the Aquatic Food Web, the output is as follows:

Tastiest Food:

Limpets
Mussels

Food Web Heights

The height of an organism in the food web is defined as the longest path from any of the producers up to that organism. Thus, all producers have height 0. Any organism that eats only producers has height 1. All other organisms in the food web have a height that is one more than the maximum height of all their prey. This is innately a recursive definition, and can be calculated using recursion. However, recursion is not required. Another fine approach is use repeated iteration as follows:

1. set all heights to 0
2. assume changes to the heights need to be made
3. visit all organisms in web[]
 - for each organism, set its height as one more than its maximum prey height
 - if no changes to the heights were made, then we are done; otherwise, redo step 3

For the Grassland Food Web example, the output is as follows:

Food Web Heights:

Grass: 0
Grasshopper: 1
Hawk: 3
Lizard: 2
Rabbit: 1
Snake: 2
Mouse: 1

Note: the organisms and their heights are printed in the order they were first entered by the user.

Categorize Organisms by Vore Type

Identify each organism in the food web as one of the following:

- **Producer:** eats no other organism in the food web; the assumption is that all producers are plants and all plants are producers.
- **Herbivore:** only eat producers (i.e. only plants)
- **Omnivore:** eats producers and non-producers (i.e. plants and animals)
- **Carnivore:** only eats non-producers (i.e. only animals)

For the Grassland Food Web example, where there are no Omnivores, the output is as follows:

Vore Types:

Producers:
Grass
Herbivores:
Grasshopper
Rabbit
Mouse
Omnivores:
Carnivores:
Hawk
Lizard
Snake

For the Aquatic Food Web example, the output is as follows:

Vore Types:

Producers:
Phytoplankton
Seaweed
Herbivores:
Limpets

```

Zooplankton
Omnivores:
    Mussels
Carnivores:
    Bird
    Crab
    Fish
    Lobster
    Prawn
    Whelk

```

Programming Task III - Species Extinction

A species has gone extinct and must be removed from the food web. The starter code already includes a scanned user-input for the organism index that has gone extinct. For example, if the user enters 3 for the extinction index, then the Lizard must be removed from the food web. The figure above shows the updated **web[]** array and updated **prey[]** array subitems for the extinction of the Lizard. Note the changes:

1. the **Org** struct with Lizard for a **name** is missing from **web[]** and all **Org** elements after Lizard have been shifted forward one index; the size of the **web[]** array is now one less
2. all organisms that had 3 (the initial index for the Lizard) in their **prey[]** arrays, which is only the Hawk in this example, now have one less element in their **prey[]** array; the index for the extinct organism is removed by shifting the remaining elements forward one;
3. additionally, all **prey[]** array elements greater than 3 must be decremented by one to account for the shift of organisms in the **web[]** array; e.g. since the Snake eats the Mouse, the Snake's **prey[]** array used to store 6, which has been updated to 5 now that the Mouse has shifted forward one index in the **web[]** array due to the Lizard's extinction.

Remember that both the **web[]** array and **prey[]** arrays are dynamically allocated, so the removal of elements requires careful freeing of some memory and manual reallocation. Do NOT use `realloc()`; instead, the reallocation should be performed manually by `malloc'`ing a new array, copy the necessary elements to the new array, freeing up the old array, and reassigning the array pointer to the new array. **Edge case: if a food web only has one organism and it goes extinct, instead of `malloc'`ing an empty array, explicitly set the pointer to NULL.**

Your task is to write the **extinction()** function, which takes in a *pointer to the web[] array*, a *pointer to the number of organisms* in the web, and the *index* for the organism to remove due to extinction. The number of organisms is passed-by-pointer since it will be changed in the function and passed back to **main()**. Similarly, the **web[]** array itself is passed-by-pointer because the array will need reallocation and may have its location changed in the function. Since **web** is an array, it is a pointer. When we pass an array name by pointer to a function, we are actually passing a *pointer to a pointer* (i.e. a *double pointer*): **Org** web**.

The starter code includes some tips for the extinction function, which are reproduced here for convenience:

```

// Remember to do the following:
// 1. remove organism at index from web[] - DO NOT use realloc(), instead...
//    (a) free any malloc'd memory associated with organism at index; i.e. its prey[] subitem
//    (b) malloc new space for the array with the new number of Orgs
//    (c) copy all but one of the old array elements to the new array,
//        some require shifting forward to overwrite the organism at index
//    (d) free the old array
//    (e) update the array pointer to the new array
//    (f) update numOrgs
// 2. remove index from all organisms' prey[] array subitems - DO NOT use realloc(), instead...
//    (a) search for index in all organisms' prey[] arrays; when index is found:
//        [i] malloc new space for the array with the new number of ints
//        [ii] copy all but one of the old array elements to the new array,
//            keeping the same order some require shifting forward
//        [iii] free the old array
//        [iv] update the array pointer to the new array
//        [v] update the numPrey subitem accordingly
//    (b) update all organisms' prey[] elements that are greater than index,
//        which have been shifted forward in the web array

```

Once the function is working properly and is fully tested, re-run all of the food web analysis steps on the updated web with the extinct species removed. At this point, your code will be tested for its full output. The full output for two input .txt files are provided here:

- [full output for GrasslandFoodWeb.txt](#) with Lizard (index 3) going extinct

- [full output for AnotherFoodWeb.txt](#) with Spider (index 10) going extinct

Notice how the Lizard's extinction has little impact on the Grassland Food Web, since it is near the top of the food web and the apex predator has many options. However, the extinction of the Spider has great impact on Another Food Web, since it is closer to the bottom of the food web and is a key food source for many organisms.

Programming Task IV - Free the Dynamic Memory

Finally, make sure to free all memory dynamically allocated to heap to prevent potential memory leaks. Dynamically allocated arrays used solely inside of functions must be freed before leaving the function. You must be VERY careful with memory management in `extinction()` to ensure all malloc'ed memory is freed. Specifically notice that you must free the memory of a `prey[]` array subitem BEFORE removing an `Org` element from `web[]`. Lastly, `main()` should end with a complete freeing of ALL `prey[]` array subitems AND the `web[]` array itself. The autograder includes memory leak checks.

Requirements

- The food web must be represented ONLY by the `Org` struct array as declared and allocated in `main()` in the starter code. No copies of `web[]` can be made, except during reallocation steps. Violations of this requirement will receive a manually graded deduction.
- Use the starter code as provided, adding code to complete functions and developing additional functions to complete the analysis steps, without making any structural changes: do NOT modify the `Org` struct definition (no name change, do not add subitems, do not remove subitems, do not modify subitem definitions, etc.), and do NOT change the function headers (no name changes, do not add parameters, do not remove parameters, do not modify parameter types, etc.). Violations of this requirement will receive a manually graded deduction.
- Solve each task and the program at large as intended, i.e. build the food web using the `Org` struct array called `web[]` and fill out the `prey[]` array subitems with indices of the `web[]` array; use this food web data structure to complete all food analysis steps. Additional arrays (static or dynamic) are allowed, as long as they don't represent the same data that is (or should be) stored in `web[]`. Violations of this requirement will receive a manually graded deduction.
- The `web[]` array and `prey[]` array subitems must be dynamically allocated to ALWAYS have the exact amount of memory needed. They `prey[]` array subitems begin empty. As predator-prey relationships are added, the individual `prey[]` array subitems should be reallocated for each additional element. **Do NOT use `realloc()`**; instead, manually malloc space for a new array with modified size, copy over necessary data, free the old array, update the array pointer, and update the size variable. During `extinction()`, both the `web[]` array and `prey[]` array subitems must be reallocated, when necessary, to utilize the exact amount of memory needed to store the food web data after the species goes extinct. **Edge case: instead of malloc'ing an empty array, explicitly set the pointer to NULL.** Violations of this requirement will receive a manually graded deduction.
- All dynamically allocated memory must be freed to prevent possible memory leaks. This issue is checked by the autograder but may also receive a manually graded deduction.
- All input `***FoodWeb.txt` files have the organisms in alphabetical order, but the predator-prey relationship are in a random order. When multiple organisms are printed, make sure the print order matches the order that the organisms and/or the predator-prey relationships were read-in from user-input. This is managed naturally by using arrays and then printing from index 0 working up in index. During the extinction of a species, make sure to shift remaining elements forward to remove the organism (i.e. do not get tricky by using a swap, just stick with shifting elements forward in the array to replace it); this keeps the order as read-in from user-input.
- Coding style issues are manually graded using deductions, worth up to 25% of the total project score. Style points are graded for following the course standards and best practices laid out in the syllabus, including a header comments, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data and control structures.
- Programming projects must be completed and submitted individually. Sharing of code between students in any fashion is not allowed. Use of any support outside of course-approved resources is not allowed, and is considered academic misconduct. Examples of what is allowed: referencing the zyBook, getting support from a TA, general discussions about concepts on piazza, asking questions during lecture, etc. Examples of what is NOT allowed: asking a friend or family member for help, using a "tutor" or posting/checking "tutoring" websites (e.g. Chegg), copy/pasting portions of the project description to an AI chatbot, etc. Check the syllabus for Academic Integrity policies. Violations of this requirement will receive a manually graded deduction, and may be reported to the Dean of Students office.

Submission

Develop your program in the IDE below. Use the "Run" button to test your code interactively as you develop your program. Use the "Submit for grading" button to test your code against the suite of autograded test cases, which also submits your

program for grading. Formally, the official code submission is your latest submission that scores the highest when run through the autograder. Late submissions are allowed for a point reduction. If you submit your highest scoring program after the deadline it will be treated as a late submission, which will incur the point reduction.

Assignment Inspiration

The overall idea for this project, food web analysis steps, and the food web data was greatly inspired by Ben Stephenson and Jonathan Hudson from University of Calgary.




Copyright Statement.



This assignment description is protected by [U.S. copyright law](#). Reproduction and distribution of this work, including posting or sharing through any medium, such as to websites like [chegg.com](#) is explicitly prohibited by law and also violates [UIC's Student Disciplinary Policy](#) (A2-c. Unauthorized Collaboration; and A2-e3. Participation in Academically Dishonest Activities: Material Distribution).


Material posted on [any third party](#) sites in violation of this copyright and the website terms will be removed. Your user information will be released to the author.


LAB
ACTIVITY


14.2.1: Food Web Analysis with Dynamic Memory

100 / 100

Run

Online (1)Aaryan Sharma

main.c

396


Submit for grading

Coding trail of your work

[What is this?](#)

9/12... T 0 W 0 | 0 | 0 | 0 | 0 | 3 | 3 R 12 | 0 | 12 | 0 | 12 | 12 | 12 | 12 | 12 | 15 | 15 | 12 | 12 | 15 | 21 | 21 | 27 | 33 | 33 F 33 | 12 | 33 | 33 | 33 | 33 | 39 | 39 | 39 | 42 | 0 | 39 | 39 S 45 | 45 | 45 | 45 | 51 | 12 | 12 | 12 | 12 | 12 | 12 | 15 | 51 | 51 | 15 | 78 | 78 | 63 | 78 | 78 | 88 | 40 | 27 | 58 | 91 | 12 | 12 | 91 | 64 | 40 | 91 M 91 | 12 | 12 | 91 | 59 | 91 | 91 | 91 T 46 | 46 | 41 | 91 | 15 | 15 | 15 | 100 ..9/19

Latest submission - 11:42 AM CDT on 09/19/23

Submission passed all tests

Total score: 100 / 100

☒ Only show failing tests (34 tests hidden)

Open submission's code

1: style - header, identifiers, comments, layout, decomposition, and data/control structures 0 / -25

see syllabus for style guidelines

This is a manual deduction.
Your instructor hasn't graded this test yet.

2: solution approach to task(s) not as intended 0 / -25

e.g. web is NOT only stored as dynamic array of Orgs, prey[] array subitems have a copy (either temporarily or always), etc.

This is a manual deduction.
Your instructor hasn't graded this test yet.

3: dynamic memory allocation/reallocation mistakes 0 / -25

e.g. extinction just removes elements by shifting the Orgs in web[] and ints in prey[] array subitems, but does not reallocate the web[] array OR the individual prey[] arrays correctly; use of realloc() is prohibited, instead malloc() the new array, copy elements from old to new, free the old, and rewire the array pointer

This is a manual deduction.
Your instructor hasn't graded this test yet.

4: structural changes to starter code 0 / -25

e.g. modification(s) to Org struct, buildWeb(), extinction(), scanf() calls, etc.

This is a manual deduction.
Your instructor hasn't graded this test yet.

5: late submission penalties 0 / -30

see syllabus for late policy






This is a manual deduction.
Your instructor hasn't graded this test yet.

6: academic misconduct 0 / -100

see syllabus for late policy

This is a manual deduction.
Your instructor hasn't graded this test yet.

5 previous submissions

11:34 AM on 9/19/23	15 / 100	View 
11:31 AM on 9/19/23	15 / 100	View 
11:24 AM on 9/19/23	15 / 100	View 
10:10 AM on 9/19/23	91 / 100	View 
10:09 AM on 9/19/23	41 / 100	View 

[Trouble with lab?](#)

[Feedback?](#)

