Learning Outcome: Students will gain experience working with dynamic memory allocations, structs, and binary tree data structures in C.

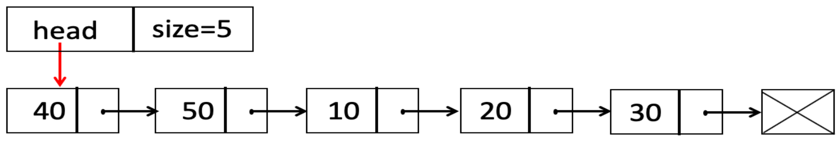
Part 1: Working with linked lists

The purpose of part 1 is to reinforce the linked list data structures you learned with additional capability. Review the linked list data structure in the lecture if needed. You will add more functionality to what you already have. Open the `linkedlist.c` file and examine it. Write the following functions.

1. `update_at(list, index, new_value)` changes the data in a node at the specified `index` to `new_value`.
2. `delete_all(list, value)` deletes all nodes that contain `value`.
3. `delete_at(list, index)` deletes a node at the specified index.
4. `print_oldest_first(list)` prints all data in the list in the order they were inserted. This function must traverse the linked list only once, which guarantees that the time complexity is of *O(n)*. You are not allowed to use additional data structures such as linked lists or arrays.
5. Test your new functions in your `main()` function.

We assume that the most recently added node is at position 0, and the node (if any) right next to it is at position 1 and so on. As an example, consider a linked list, named *list,* contains five data values added in the following order:

```
add(list, 30), add(list, 20), add(list, 10), add(list, 50),
add(list, 40)
```



- `print_oldest_first(list)` should print 30, 20, 10, 50, 40 in order.
- `update_at(list, 1, 500)` should change the data in the node at position 1 from 50 to 500 and return 1, indicating a success.
- `update_at(list, 5, 100)` should return 0, indicating an error.
- `delete_at(list, 3)` should delete the node containing 20 and return 1.
- `delete_at(list, 5)` should return 0.
- `delete_all(list, 40)` should delete all nodes containing 40, if any.

Part 2: Password Hunting

Your mission in part 2 is to search a password hidden somewhere in your computer memory. The password is a ten-digit decimal number stored in the heap section. The number is so important and secrete, so it had been split into two halves, each of which has been buried along with many other similar-looking numbers. Please do not disassemble your computer to find it. Instead, thoroughly follow the instructions given below.

Main Steps

1. Step 1 – Find the memory address where the integer value 78405 is stored.
2. Step 2 – Compute the difference, called `offset`, between the address you found in step 1 and the start address of the heap. Get the last two hexadecimal digits of `offset` and convert it as a binary number which we will call `path`. For example, if `offset` is `0x17A`, the `path` will be "`01111010`".
3. Step 3 – Sort the heap
4. Step 4 – Construct a binary search tree containing all sorted elements in the heap.
   a. The `make_bst` function is provided for you to use in this step.
   b. From this point forward, the values on the heap will no longer be needed. So free the memory.
5. Step 5 – Search the tree.
   a. The `search_bst` function is provided for you to use in this step.
   b. The first half of the password is in a tree node that you can reach by following `path` you computed in step 3. The letter '0' in the path means 'go to the left child' and '1' means 'go to the right child'. Using the path "`01111010`" as an example, your search starts from the root of the tree and goes down as follows: left ('0'), right ('1'), right ('1'), right ('1'), right ('1'), left('0'), right('1'), and then left('0').
   c. The second half of the password is at the 6th node from the left in the bottom of the tree. You will need to construct another path for this manually.

Before you start step 1

Open and study `binary_search_tree.h` and `binary_search_tree.c` files.

1. Define a new function named `void inorder_traversal(struct treenode *current)`, which visits all tree nodes in order (i.e., left subtree, root, right subtree) printing the value in the `current` node.
2. Create a `main` function and experiment with the `make_bst`. You may create a small sorted array of integers such as {10,20,…, 150} and use it to create a binary search tree. Use your `inorder_traversal` function to make sure that the tree has been constructed correctly. You should be able to see all numbers in ascending order.
3. Experiment with `search_bst`. Find values in the tree following a path. Use different paths such as "0", "1", "00", "01", "10", …, "111".
4. Once you understand how the functions work, comment out your `main` function in the file.

Additional Instructions

1. Step 1 – finding the location of 78405
   - If the number exists in more than one place, you need to use the highest address containing the value.
2. Step 3 – heap sorting
   - There are 1023 integer values stored in contiguous heap memory locations.
   - Use the following algorithm to sort them:
     - Find the minimum value and swap it with the element at the start position (i.e., the lowest address) where the first element is stored.
     - Find the second minimum value and swap it with the element at the second lowest memory, and so on.
     - Repeat the swap operations until all the elements are sorted.
3. Note that the heap we are talking about here is not the data structure heap you might learn in some other programming courses. So, googling "heap sort" will only confuse you.
4. The values in the heap will be stored in a binary search tree. You are NOT allowed to use any other data structures. For example, you will not create an array to sort the heap or construct a binary tree.

Requirements

1. Add the following functions to the `password_hunter.c` file.
- `void sort_heap(int *begin)` – sorts integer values stored in the heap location starting at `begin` address. Do not create any data structures. You will only work with pointers using minimum memory required to sort values.

- `int *get_address(int target, int *begin)` – returns the address of `target` in the heap memory starting at `begin`. If more than one `target` exits, returns the highest address containing target. If `target` does not exist, returns NULL.

- `unsigned long get_offset(int *begin, int *middle)` – returns the difference between `begin` and `middle`. Note that the type of any pointers is essentially `unsigned long`. Any pointers can be converted to it:

```
unsigned long var = (unsinged long) pointer
```

- `int main()` – prints the following information in order:
  - the address of the target element (78405)
  - the offset as a hexadecimal format, i.e., the target address relative to the start address (before sorting).
  - the path you used to search the first half of the password
  - the path you used to search the second half of the password
  - the entire password

2. You should also write the `inorder_traversal` and the commented `main` function in the `binary_search_tree.c` file.
3. You are not allowed to use any data structures other than a binary search tree to store heap elements.

Optional helper functions

1. `char *get_path(unsigned long offset)` – returns an 8-bit string representing the last two hexadecimal digits in `offset`. If you are not writing `get_path`, you should compute the `path` manually and use the hardcoded string literal in your `main` function.
2. You may define any other helper functions if you want.

Compile and run

- `gcc -o password_hunter binary_search_tree.c password_hunter.c`
- `./password_hunter`
- Your program must generate the following output:

```
address of 78405 is 0x[redacted]
offset is [redacted]
path1 = [redacted]
path2 = [redacted]
Password is [redacted]
```

Deliverables

1. You must upload your final solutions `linkedlist.c` `binary_search-tree.c` and `password_hunter.c` to your Assignment 04 repository before 9/22/2021 at 7 PM.
2. You must get sign-off from one of the instructors or TAs during their office hours before the deadline.

Sign offs – Each signature is worth 1/N of your lab grade, where N is the number of signatures

- The student could update values stored in a linked list and print elements backward without using an additional data structure.

- The student implemented two versions of delete functions correctly.

- The student could sort heap and compute the location gap between the target value and the first value in the heap.

- The student could use the provided functions `make_bst` and `search_bst` to construct a binary search tree and search two password parts.

- The student could print all the information in the `main` function: the address of the target, offset, two paths, and the password. Note that the target address may vary.