Assignment 3

November 6, 2024

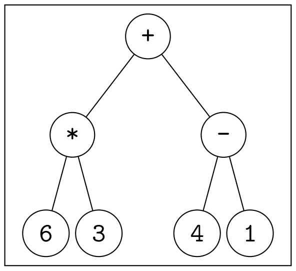
The assignment is due on **15 November, 2024, 9:00 PM** Late submissions will be accepted until 18 November, 2024, 9:00 PM with a penalty of 10%. After that no submissions will be accepted.

1 Written Questions

Question 1: (20 points)

An important application of binary trees is to represent the structure of an arithmetic computation.

Example Expression Tree: 6 x 3 + (4 - 1)



a) Sketch the following binary expression trees:

1. $\frac{3-2}{0.5} \times ((9-5)+2) - (7 \times \frac{3}{4} + 9)$
2. $(\frac{3}{2})^5 \times (N - M \times 2^M) - (7 \times 12 + 9)$

b) Explain how the order of operations are executed in terms of the levels in an arithmetic expression tree.

Hint: Think of the first and last operations executed in an arithmetic expression tree.

2 Coding Questions

Question 1 (35 Points)

Overview: Trees are an important non-linear data structure that allows us to express many different kinds of structural and hierarchical relationships. Although a binary tree is one of the more common types of trees you'd encounter, it is important to note that a tree's nodes can have an arbitrary number of children.

Your task here is simple: given an array-based representation of a *general* tree, construct a tree from the input using the provided `TreeNode` class. You will write the function `read_tree()` that will read the general tree from standard input and return the root of the tree as a `TreeNode` object.

The first line of the input will provide `n` and `d`. `n` represents the length of the array-based representation and `d` represents the maximum number of children a node can have. Not all nodes will have `d` children and so the dash character `-` represents an empty child.

The second line of the input is a list of `n` space-separated strings that represents the tree as an array. Each string represents a page/node in the tree and the dash character (`-`) represents an empty child and should be ignored.

You will be provided a `TreeNode` class in a separate file `treenode.py`. This file will be imported in your solution and its notable methods are:

- `a = TreeNode('A')`: Constructs a `TreeNode` with a value of `'A'` and store it in a variable `a`.
- `a.get_value()` returns the value of the node `A`.
- `a.get_children()` returns a list of `TreeNode` objects that are the children of the node `a`.
- `a.add_child(child)` adds a `TreeNode` object, `child`, to the list of `a`'s children.
- `a.print_tree()` prints the tree to standard output. Each line in the output will have the following format: `NODE -> CHILDA CHILDB CHILDC ...`

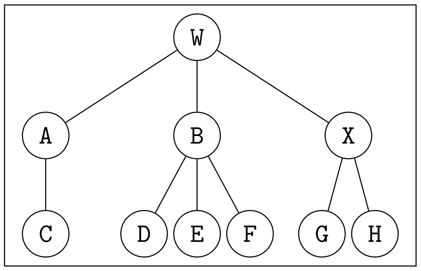
Example Input #1:

```
12 3
W A B X C - - D E F G H
```

Example Output of `read_tree().print_tree()`:

```
W -> A B X
A -> C
B -> D E F
X -> G H
```

Example Tree #1:



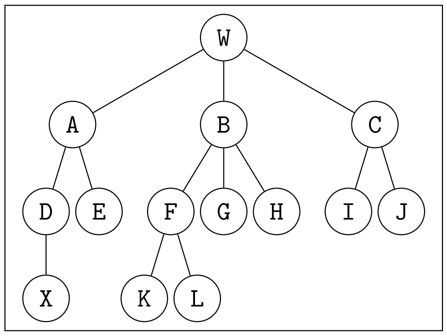
Example Input #2:

```
24 3
W A B C D E - F G H I J - X - - - - - - - - K L
```

Example Output of `read_tree().print_tree()`:

```
W -> A B C
A -> D E
D -> X
B -> F G H
F -> K L
C -> I J
```

Example Tree #2:



Specifications:

- A node's value can either be a single character (eg. `A`) or a sequence of characters (eg. `AA`, `AB`).
- At least one node will have `d` children.

- Children should be added to a node **in the order they appear in the input**. Input should also be processed from left to right.
- When the second line of the input ends, you should stop adding nodes to the tree.
- The dash character `-` indicates an empty child and should not be added as a node.
- **Do not modify the `main()` function.**
- **Do not modify the provided `treenode.py` file**. You also do not need to submit the `treenode.py` file.

Marking Rubric: This question will be marked out of 35 for correctness (pass test cases):

- 35/35: Pass all 10 of the test cases
- 21/35: Pass any 6 (or more) of the test cases
- 14/35: Pass any 3 (or more) of the test cases
- 7/35: Pass any of the test cases

Question 2 (35 points)

Overview: In this question6, your task will be to implement a few methods of the class `LinkedList` which implements the linked list data structure. Use the existing methods of the `LinkedList` class to help implement the required methods.

You will be provided a `LinkedList` class. The class includes various methods you to help you complete the assignment:

- `llist = LinkedList()`: Constructs a `LinkedList` object. Each linked list is constructed of Node() objects.
- `llist.head` returns the pointer to the head node of the linked list.
- `llist.insertAtBegin(self, data)` adds a node to the beginning of the linked list with a value of `data`.
- `llist.insertAtIndex(self, data, index)` adds a node at `index`. Indexing starts from 0.
- `llist.insertAtEnd(self, data)` adds a node to the end of the linked list with a value of `data`.
- `llist.updateNode(self, val, index)` updates the node at `index` to the value `val`.
- `llist.sizeOfLL(self)` returns the size of the linked list (number of nodes).

You will be provided a `Node` class. The node class only contains two attributes and no methods. Node objects construct a linked list.

- `node = Node()`: Constructs a `Node` object.
- `node.data` returns the value stored at the current node.
- `node.next` returns the pointer to the next node in the linked list.

Hint: You can create new pointers to nodes of the linked list. For example, `current_node = self.head` will create a new pointer to the head of the linked list object, which you can use to iterate through the linked list.

Remove Duplicates

Description: Implement the method `remove_duplicates()` that removes all duplicate nodes from a singly linked list while preserving the original order of the first occurrence of each value. A node is considered a duplicate if its value matches any previously seen values in the linked list.

Examples:

```
llist = 3 -> 2 -> 0 -> -4 -> 2
result:  3 -> 2 -> 0 -> -4

llist = 1 -> 2
result:  1 -> 2

llist = 6 -> 5 -> 6 -> 7
result:  6 -> 5 -> 7

llist = 1 -> 1 -> 2
result:  1 -> 2

llist = 4 -> 3 -> 2 -> 3 -> 1 -> 2
result:  4 -> 3 -> 2 -> 1
```

Specifications:

- The linked list can contain any number of duplicates
- The first occurrence of each duplicate value should be preserved
- $1 \leq len(llist) \leq 100$
- The linked list nodes should remain in the same relative order after duplication removal (i.e., no sorting)
- The linked list node values will consist of integers only
- You **cannot** call the Node() constructor or the LinkedList() constructor to instantiate new objects. You must instead create new pointers to nodes in the linked list and reassign pointers in the linked list.
- You can create any new methods in the linked list or node class.

Merge

Description: Implement the method `merge(self, llist2)` that merges two sorted singly linked lists together. The method must merge the nodes in place. This means that the reversal must be performed within the existing linked list structure without instantiating any new node or linked list objects. The result must add all the nodes from the `llist2` object that is passed to the method into the linked list the method is called on. The resulting merged linked list must maintain the sorted order after merging.

Examples:

```
llist1 = 1 -> 3 -> 5
llist2 = 2 -> 4 -> 6
Result:  1 -> 2 -> 3 -> 4 -> 5 -> 6

llist1 = 2
llist2 = 1 -> 3
Result:  1 -> 2 -> 3

llist1 = 1 -> 8
llist2 = 1 -> 4 -> 6
Result:  1 -> 1 -> 4 -> 6 -> 8
```

Specifications:

- If one list is longer than the other, append the remaining nodes of the longer list at the end.
- Both linked lists will have at least one node.
- Both linked lists are sorted in ascending order.
- Both linked lists only contains integers for their data values.
- You **cannot** call the Node() constructor or the LinkedList() constructor to instantiate new objects. You must instead create new pointers to nodes in the linked list and reassign pointers in the linked list.
- You can create any new methods in the linked list or node class.

Marking Rubric: This question will be marked out of 35 for correctness (pass test cases):

- 35/35: Pass all 20 of the test cases
- 21/35: Pass any 15 (or more) of the test cases
- 14/35: Pass any 10 (or more) of the test cases
- 7/35: Pass any of the test cases

3 Writing and testing your solution

For the programming questions, you should edit the provided files and add your solution. You can check your solution by running `driver.py`. You can see the output of your program in the `Output/` folder, and any errors in the `Error/` folder.

Please do not change the driver file, only edit the solution file.

You can run the driver in a terminal as follows (assuming your working directory is the assignment folder):

```
$ cd assignment1
$ cd Question1
$ python3 driver.py
All tests passed!
$
```

Please add your name, student number, ccid, operating system and python version at the top of each solution file by replacing the provided comments.

4 Submission Instructions (10 points)

Please follow these submission instructions carefully. Correctly submitting all files is worth 10 points. In these files, you must replace `ccid` with your own ccid. Your ccid is the first part of your UAlberta email (ccid@ualberta.ca). Do not zip any of these files. Please submit the following files to eclass:

- `ccid_writtenQuestions.pdf` : Your answers to all written questions should be in this one pdf file.
- `ccid_solutionQuestion1.py` : Edit the provided file for question 1. After your solution passes all test cases (which you must test using the driver), rename the file to include your ccid, that is, `ccid_solutionQuestion1.py` and submit it to eclass.
- `ccid_solutionQuestion2.py` : Edit the provided file for question 2. After your solution passes all test cases (which you must test using the driver), rename the file to include your ccid, that is, `ccid_solutionQuestion2.py` and submit it to eclass.