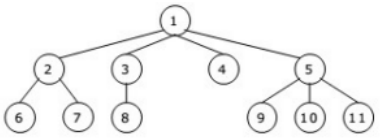
Procesamiento de un archivo de formato JSON

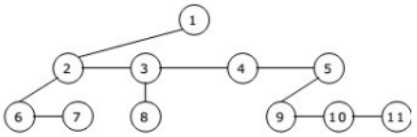
AyED TP - 2

Para este trabajo se debe realizar el procesamiento de un archivo del tipo .json que contiene información sobre los laureados con un premio nobel.
La estructura de un archivo tiene distintos niveles de profundidad con múltiples objetos en cada nivel, por lo que este formato puede verse como un arbol m-ario.
La lectura del archivo debe cargarse en memoria sobre una estructura de un arbol m-ario.
Este tipo de árbol debe implementarse sobre un árbol binario. La forma de implementar un arbol m-ario sobre un árbol binario es la siguiente:
La raíz del arbol m-ario es la misma que la del binario. El hijo más hacia la izquierda del árbol m-ario es el izquierdo del binario. Los restantes hijos del m-ario (hermanos del izquierdo), se implementan como una cadena de hijos derechos del primer hijo.

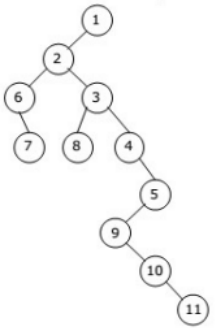


Solution:

Stage 1 tree by using the above mentioned procedure is as follows:



Stage 2 tree by using the above mentioned procedure is as follows:



Deberá implementar de esta forma un arbol m-ario sobre un árbol binario. Sobre esta implementación la inserción de hijos (hermanos) deberá ser de forma tal que permita reordenarlos una vez cargada toda la información y alternativamente cargarlos en forma ordenada, es decir, cada nuevo nodo que sea hermano de otro (insertado sobre la cadena de la rama derecha), debe quedar insertado en forma ordenada creciente respecto de algún valor que tenga dicho objeto (nombre, país, fecha nacimiento, etc).
Debera implementar un recorrido pre-orden en el m-ario para poder listar el resultado de la carga y ordenamiento.

Resolución

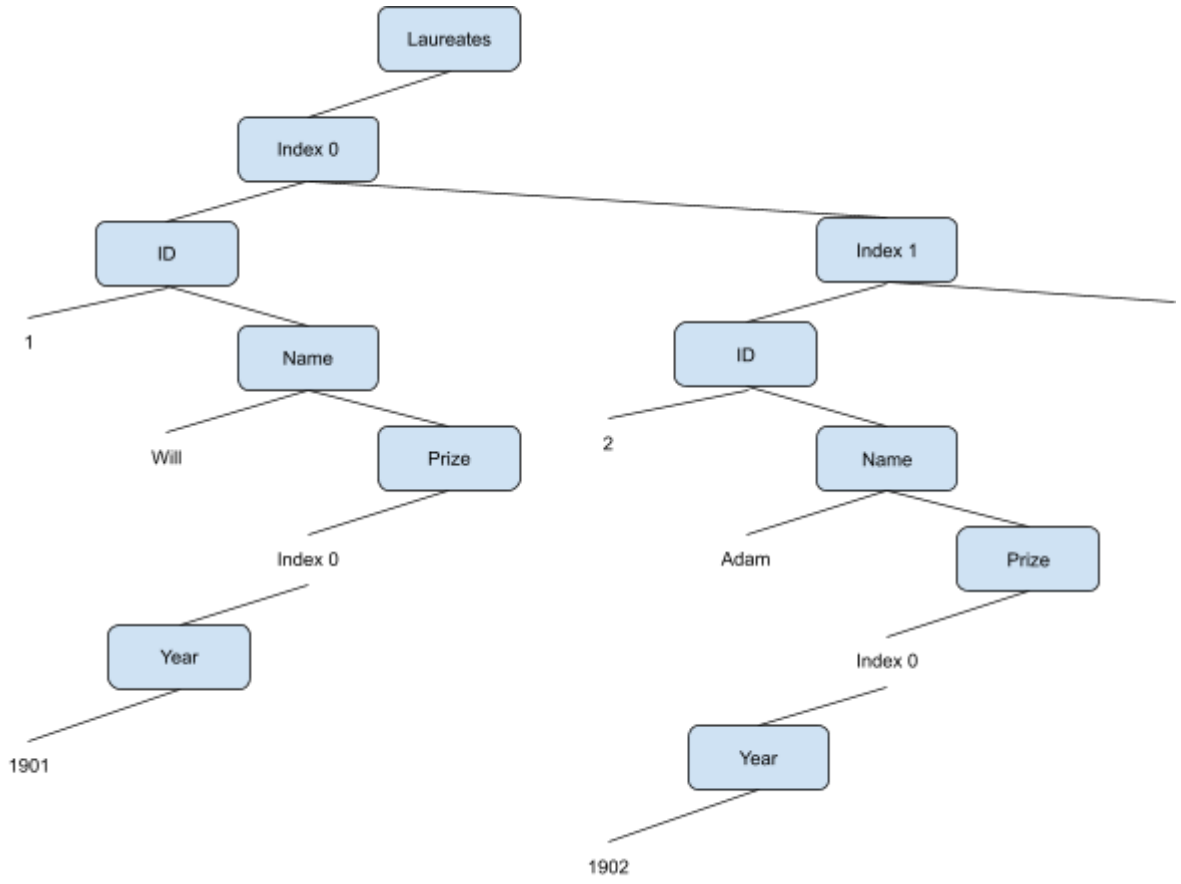
Para realizar el programa se reutilizaron las clases y códigos de nodos y arboles utilizados y estudiados durante la cursada. También se utilizó la librería de manejo de json de nlohmann para ordenar y parsear este archivo.

1. Se inicia leyendo el archivo JSON utilizando la librería nlohmann, guardando esto en la variable **j** de manera ordenada, este objeto contendrá la estructura y los datos del JSON.
2. Creamos dos vectores "**serial**" y "**vectorToOrder**", donde el primero almacena los datos serializados del JSON mientras que el otro se utilizará para almacenar los datos que se ordenarán posteriormente.
3. Creamos dos objetos de la clase "**Tree<obj>**" llamados **T1** y **T2**, donde se guardará el árbol binario original y el árbol binario ordenado respectivamente.
4. Llamamos a la función ***jsonToVector()*** para indexar y organizar el JSON, se pasa el objeto **j** que corresponde al json, el vector serializado "**serial**" y un valor de jerarquía.
5. Iniciamos la fase con el usuario, preguntando la manera en la que se quiere ordenar el JSON.
6. Si la respuesta es *default* se llama a la función ***createBinaryTree()*** pasando el vector serializado y el la referencia a **T1**. Esta función crea un árbol binario a partir del vector serializado y lo almacena en **T1**. Luego se muestra este árbol binario utilizando el método ***showTree()***.
7. Si la respuesta es por *key*, luego se pregunta por qué tipo de clave hay que organizar, se llama a función ***createBinaryTree()*** pasando el vector serial y la referencia a T1. Llamamos a la función ***sortTree()*** pasando la key, el vector serial, el vector a ordenar y el árbol T1. Esta función ordena el árbol binario utilizando el método de inserción y devuelve el nuevo árbol ordenado T2.
8. Una vez finalizado el programa, se muestran los datos ordenados y finaliza el programa.

JSON to binary Tree.

Al iniciar el programa nosotros contamos con un archivo JSON el cual se puede pensar como un árbol m-ario, donde cada nodo puede tener hasta m hijos, y cada hijo está asociado con una etiqueta o clave que lo identifica. De manera similar, en un JSON, los objetos se representan como pares de key-value, donde la key es un identificador y el value puede ser un objeto, un array, un string, un número, un booleano o nulo. Los objetos JSON pueden anidarse, lo que significa que pueden contener otros objetos JSON como valores de sus keys.

De esta manera, cada objeto en el JSON se puede considerar como un nodo en el árbol m-ario, y las keys de los objetos son las etiquetas que identifican a cada hijo. Los arrays en JSON también se pueden ver como nodos hijos en un árbol m-ario, donde los elementos del array se corresponden con los hijos del nodo padre.



Para crear este árbol se utiliza la lógica mencionada en la consigna, luego para insertar estos nodos en el árbol se tiene en cuenta si son un *“Array” ; “Index” ; “Key” o “Value”.* Para ello dentro del método ***searchTree()*** se compara con condicionales y de manera recursiva el nodo a insertar con el puntero al nodo actual.

Los nodos se trabajan como un tipo T, donde aparte de tener punteros hacia el nodo izquierdo y derecho, se tiene una variable info del tipo T la cual es un obj con el tipo de dato, el dato y el nivel de jerarquía en el vector. Este objeto está declarado en un struct.

Ordenamiento del árbol.

Para ordenar el árbol se parte desde serializar el JSON, luego pasar por el algoritmo nombrado anteriormente de convertir el árbol m-ario a árbol binario cómo el grafico, pero se tiene un paso extra donde se pregunta la key por el que se quiere ordenar, una vez que se obtiene la key con ella, el vector serializado y el árbol binario formado se forma un nuevo vector con los valores de las key buscadas, a este vector se lo pasa por un algoritmo de inserción para ordenarlo, y se vuelve a formar el árbol a partir de este nuevo vector.

El método de ordenamiento ***sortTree()*** tiene como objetivo construir un árbol binario ordenado **“T1”** a partir de un árbol original desordenado **auxTree** y un vector de objetos **objOrd** (**vectorOrd**) que contiene información sobre las ramas del árbol original, ordenadas de acuerdo con una clave específica **key.**

La función comienza inicializando el árbol ordenado **T1** y luego utiliza el método ***vectorData()*** para recorrer el árbol original **auxTree** y llenar el vector **vectorOrd** con objetos **objOrd**, que contienen punteros a los nodos del árbol original y el valor asociado a la clave **key** de cada nodo.

Luego, se utiliza el algoritmo de ordenamiento ***insertionSort()*** para ordenar los elementos del vector **vectorOrd** en función del valor de la clave **key**.

A continuación, se crea el árbol ordenado **T1** mediante el método ***createSearchTree()***, utilizando el primer elemento del vector **vectorJson** (que contiene los elementos del árbol original convertidos en un vector) para construir el nodo raíz del árbol ordenado.

Después, se realiza un proceso de ajuste de punteros para vincular los nodos del árbol ordenado en función del orden especificado por el vector **vectorOrd**. Esto se realiza mediante dos ciclos for. El primer ciclo limpia los punteros derechos de los nodos del árbol ordenado **T1**, mientras que el segundo ciclo establece correctamente los punteros derechos para anidar los nodos en el árbol ordenado.

Finalmente, se vincula el nodo raíz de **T1** con la estructura de punteros del árbol ordenado **T1**, asegurando que la estructura del árbol esté correctamente conectada de acuerdo con el orden especificado por **vectorOrd**.

En resumen, el árbol original **auxTree** no se modifica durante el proceso de ordenación, este se utiliza para obtener información sobre la estructura de datos original. El vector **vectorOrd** se utiliza para almacenar información sobre los nodos del árbol original y sus valores asociados a la clave key, y se utiliza para determinar el orden de los nodos en el árbol ordenado **T1**. El vector **vectorJson** se utiliza para crear el árbol **T1** en función de la estructura del JSON original, pero la información de ordenación proviene del **vectorOrd**, no del **vectorJson**.