ntree_insert.c, path_exists.c, and ntree_free.c

github path for
files:
https://github.com/johndspence/holbertonschool-low_level_programming/tree/master/n_trees

Logic and Code Review

Holberton School

20160816

John Spence

ntree_insert.c Task

0. Insert mandatory

Done!

Who is done?

Help!

Write a function that insert a node in a N-ary tree.

- Prototype: `int ntree_insert(NTree **tree, char **parents, char *data);`
- `tree` is a pointer to the address of the root node. If it's `NULL`, the Tree is empty, so the node to insert will become the root node.
- `parents` is an array of string.
    - The string at the index `i` will be the content of one of the node of the tree at the depth `i`. You can assume it, there will be no trick.
    - It will always be `NULL` terminated, you can assume it.
    - If `tree` points to `NULL` (the tree is empty), `parents` can either be `NULL` or an array with the first and only element equals to `NULL`.
    - The first element will be the content of the root node.
    - The second element will be the content of one of the children of the root node, and so on ...
    - Each element of `parents` will be present in the Tree, you don't have to worry about that.
- `data` is the string to duplicate and to store inside the new node.
- The way you add an element to a `List` of children doesn't matter. You can either add it at the beginning or at the end of the `List`.
- The function should return `0` on success, or `1` if something went wrong.
- You are allowed to use `strdup`, and `strcmp` (and `malloc`, of course).

If you want to test your function with the following example, you can download the following files:

- string_split - Prototype of the function: `char **string_split(const char *string, char separator);`
- free_str_array - Prototype of the function: `void free_str_array(char **array);`
- ntree_print - Prototype of the function: `void ntree_print(NTree *tree);`
- ntree_free - Prototype of the function: `void ntree_free(NTree *tree);`
- These files are object files. That means that they are compiled. You can link them with your `main` and other source files to form an executable.
- These files are compiled with the `-g` gcc flag.

ntree_insert.c Task: Example main.c

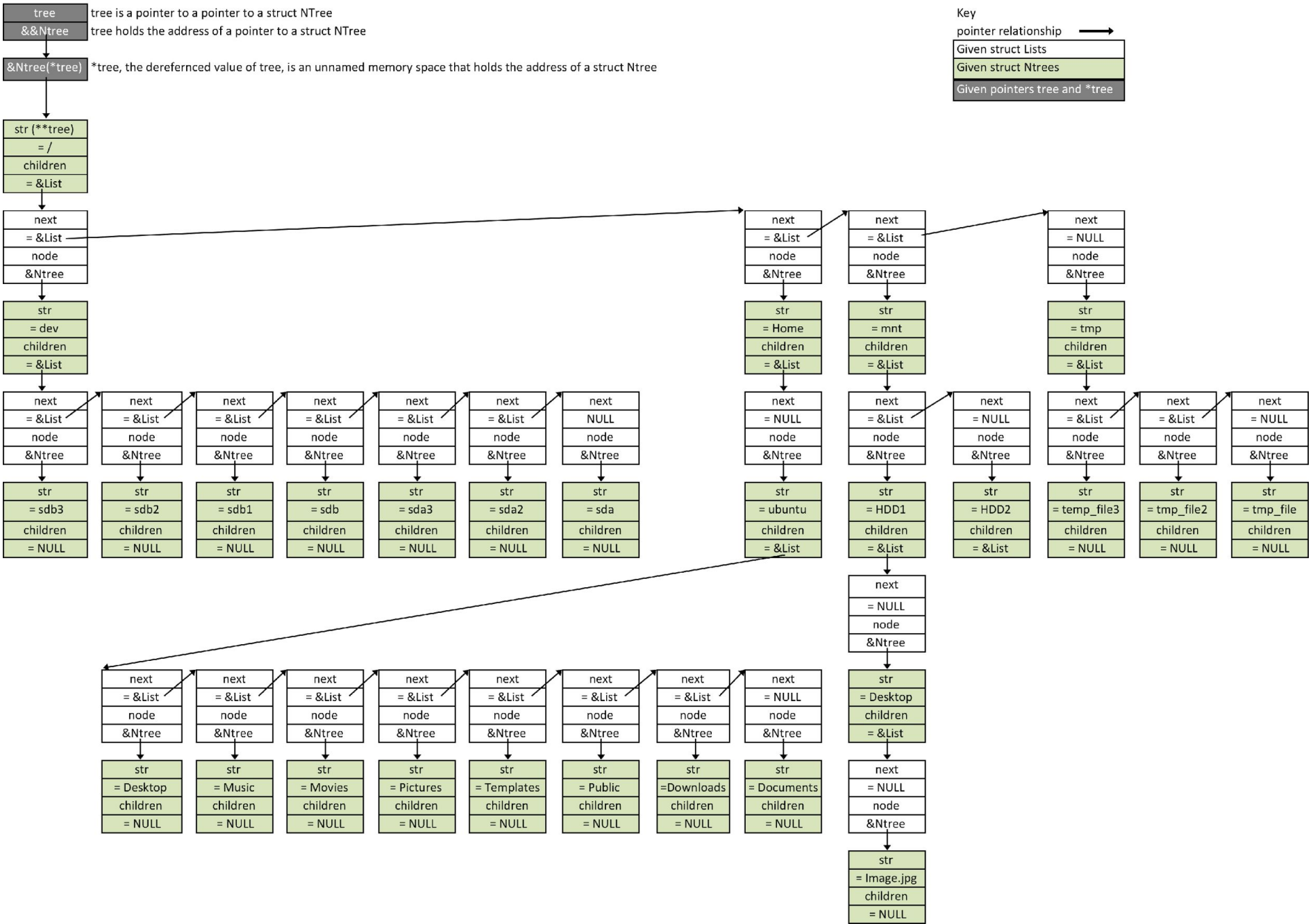
In the following example, we build a N-ary tree with something familiar: folder/files tree:

```
alex@ubuntu:/tmp/ntrees$ cat main.c
#include <stdlib.h>
#include "tree.h"

int ntree_insert(NTree **, char **, char *);

void ntree_print(NTree *);
void ntree_free(NTree *);
char **string_split(const char *, char);
void free_str_array(char **);

int main(void)
{
    NTree *tree;
    char **array;

    tree = NULL;
    ntree_insert(&tree, NULL, "/");

    ntree_insert(&tree, (array = string_split("/", ' ')), "tm
p");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ tmp", ' ')),
 "tmp_file");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ tmp", ' ')),
 "tmp_file2");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ tmp", ' ')),
 "tmp_file3");
    free_str_array(array);

    ntree_insert(&tree, (array = string_split("/", ',')), "mn
t");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ mnt", ' ')),
 "HDD1");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ mnt HDD1", '
')), "Desktop");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ mnt HDD1 Desk
top", ' ')), "image.jpg");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ mnt", ' ')),
 "HDD2");
    free_str_array(array);

    ntree_insert(&tree, (array = string_split("/", ',')), "hom
e");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home", ' ')),
 "ubuntu");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu",
 ' ')), "Documents");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu",
 ' ')), "Download");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu",
 ' ')), "Public");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu",
 ' ')), "Templates");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu",
 ' ')), "Pictures");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu",
 ' ')), "Videos");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu",
 ' ')), "Music");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu",
 ' ')), "Desktop");
    free_str_array(array);

    ntree_insert(&tree, (array = string_split("/", ',')), "de
v");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ dev", ' ')),
 "urandom");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ dev", ' ')),
 "sda");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ dev", ' ')),
 "sda1");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ dev", ' ')),
 "sda2");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ dev", ' ')),
 "sdb");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ dev", ' ')),
 "sdb1");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ dev", ' ')),
 "sdb2");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ dev", ' ')),
 "sdb3");
    free_str_array(array);

    ntree_print(tree);
    ntree_free(tree);
    return (0);
}
```

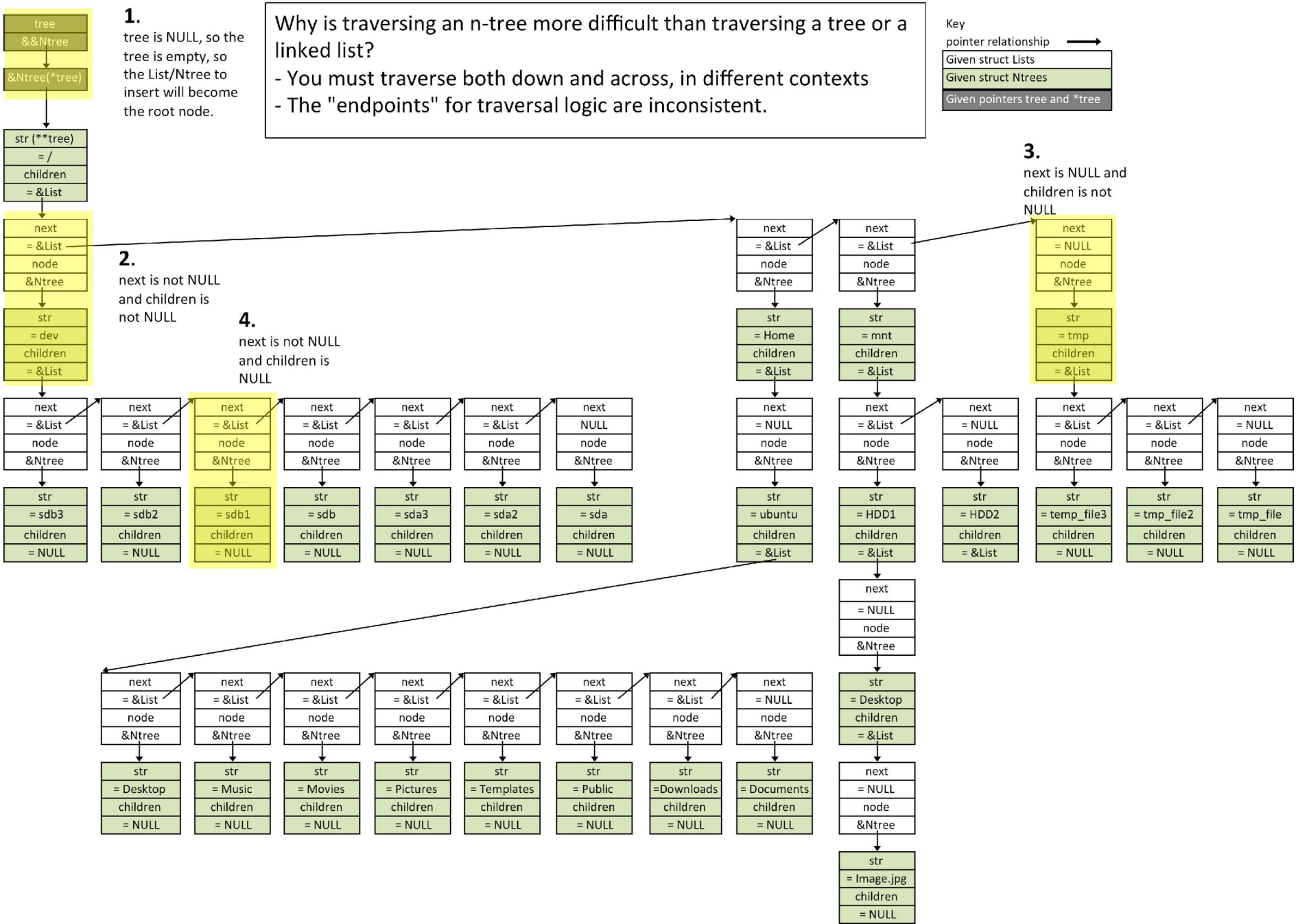
ntree_insert.c Task: Expected Output

```
alex@ubuntu:/tmp/ntrees$ gcc -Wall -Wextra -Werror -pedantic -
g main.c ntree_insert.c ntree_print.c ntree_free.c string_spli
t.c free_str_array.c
alex@ubuntu:/tmp/ntrees$
alex@ubuntu:/tmp/ntrees$
alex@ubuntu:/tmp/ntrees$ valgrind ./a.out
==55611== Memcheck, a memory error detector
==55611== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Se
ward et al.
==55611== Using Valgrind-3.10.1 and LibVEX; rerun with -h for
 copyright info
==55611== Command: ./a.out
==55611==
/
  dev
    sdb3
    sdb2
    sdb1
    sdb
    sda2
    sda1
    sda
    urandom
  home
    ubuntu
      Desktop
      Music
      Videos
      Pictures
      Templates
      Public
      Download
      Documents
  mnt
    HDD2
    HDD1
      Desktop
        image.jpg
  tmp
    tmp_file3
    tmp_file2
    tmp_file
==55611==
==55611== HEAP SUMMARY:
==55611==     in use at exit: 0 bytes in 0 blocks
==55611==   total heap usage: 177 allocs, 177 frees, 2,065 byt
es allocated
==55611==
==55611== All heap blocks were freed -- no leaks are possible
==55611==
==55611== For counts of detected and suppressed errors, rerun
 with: -v
==55611== ERROR SUMMARY: 0 errors from 0 contexts (suppressed:
 0 from 0)
alex@ubuntu:/tmp/ntrees$
```

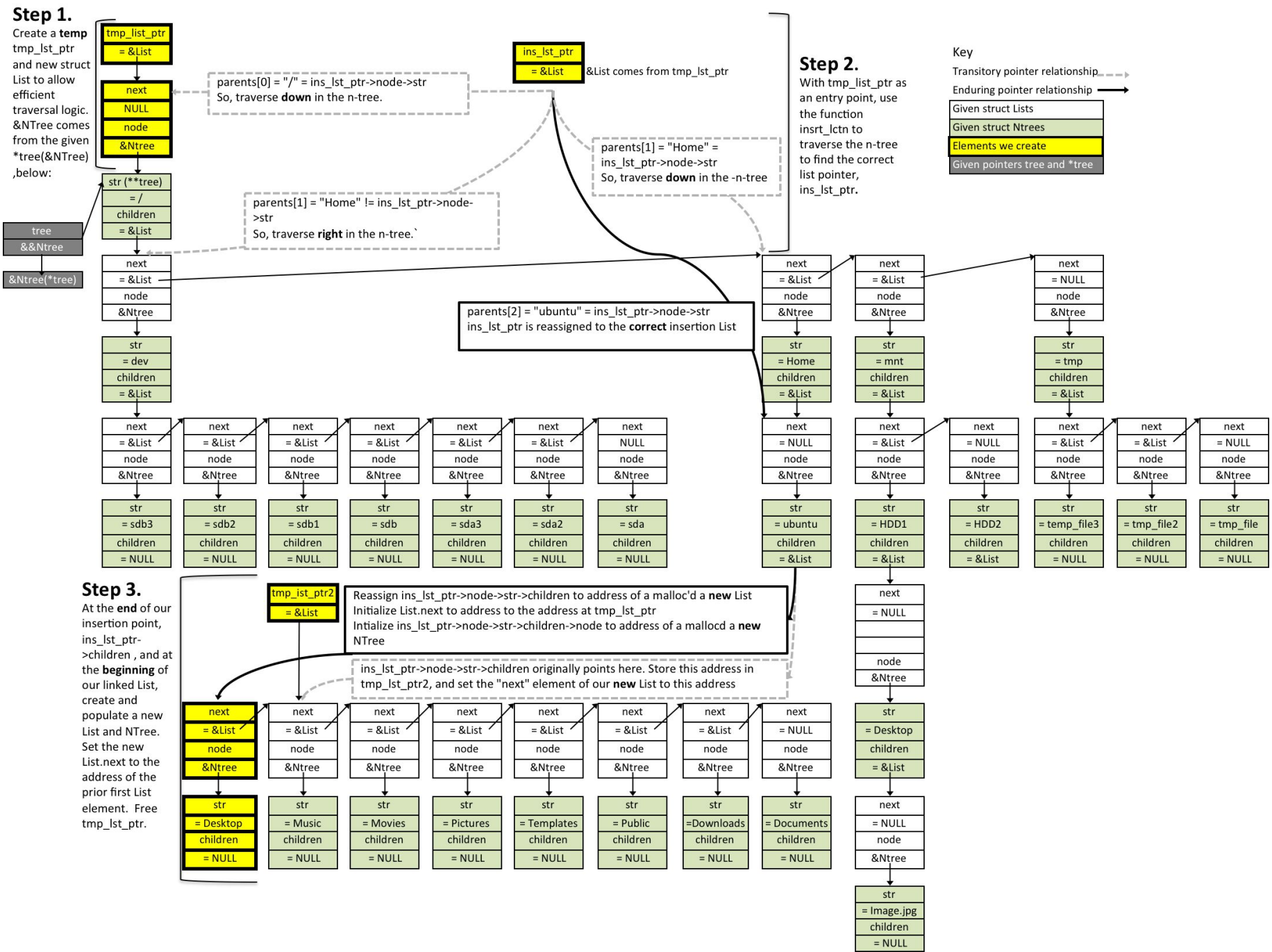
Complete n-tree



4 Scenarios that ntree_insert traversal logic must address



Insert a new node into the n-tree using : ntree_insert(&tree, (array = string_split("/ home ubuntu", ' ')), "Desktop");



Insert a new node into the n-tree using : ntree_insert(&tree, (array = string_split("/ home ubuntu", ' ')), "Desktop");

Step 1

```c
int ntree_insert(NTree **tree, char **parents, char *data)
{
    List *ins_lst_ptr;
    List *tmp_lst_ptr;
    List *tmp_lst_ptr2;

    /*
     * If tree holds NULL, there is no root NTree; create it and populate it's
     * str element with data.
     */
    if (*tree == NULL)
    {
        *tree = new_ntree(data);
        tree = &(*tree);
        return 0;
    }
    else
    {
        /*
         * Create a temporary pointer to a struct List and the corresponding
         * (also temporary) struct List itself. Set the node element of the
         * struct List to point to the first struct NTree by reusing the pointer
         * address held at *tree. This temporary pointer and struct used to
         * provide a consistent pattern upon which to design the tree traversal
         * logic
         */
        tmp_lst_ptr = new_list();
        tmp_lst_ptr->next = NULL;
        tmp_lst_ptr->node = *tree;
        /*
         * Find the correct struct List and struct NTree, under which we insert
         * the new List and Ntree. Do not modify tmp_lst_ptr, because we will
         * use it later to free the temporary NTree.
         */
        ins_lst_ptr = insrt_lctn(tmp_lst_ptr, parents);
        /* Create a new list and node and populate the node with str */
        tmp_lst_ptr2 = ins_lst_ptr->node->children;
        ins_lst_ptr->node->children = new_list();
        ins_lst_ptr->node->children->node = new_ntree(data);

        /* Insert new node at beginning of linked list */
        ins_lst_ptr->node->children->next = tmp_lst_ptr2;
        free(tmp_lst_ptr);
    }
    return 0;
}
```

The basic logic:

- If there is not a List/NTree, create it.
- Create a temporary list pointer, List, and NTree.
- Iterate through the parent elements and traverse the tree.
- If you find parents[i] in the tree, go down the tree.
- If you dont find parents[i] in the tree, go right in the tree.
- Repeat

```c
List *insrt_lctn(List *ins_lst_ptr, char **parents)
{
    int array_len;
    int i;

    array_len = array_length(parents);
    i = 0;
    /*
     * Iterate through parents/path, except for the last element of parents/
     * path. When we exit this while loop, we have the correct struct List and
     * struct NTree for the second to the last element of the parents/path.
     */
    while (i < array_len - 1)
    {
        if (strcmp(ins_lst_ptr->node->str, parents[i]) == 0)
        {
            ins_lst_ptr = ins_lst_ptr->node->children;
            i++;
        }
        else
        {
            ins_lst_ptr = ins_lst_ptr->next;
        }
    }
    /*
     * parents[i] is now the final element in the array.  We handle this element
     * separately because the logic above depends on "children" not being NULL
     * and the final element of our path may have children equal to NULL
     */
    while (strcmp(ins_lst_ptr->node->str, parents[i]) != 0)
    {
        ins_lst_ptr = ins_lst_ptr->next;
    }
    return ins_lst_ptr;
}
```

path_exists.c Task

1. Path exists mandatory

Done!

Who is done?

Help!

Write a function that check if a path is present in a N-ary tree.

- Prototype: `int path_exists(NTree *tree, char **path);`
- `tree` is the address of the root node of the Tree.
- `path` is an array of string.
    - The string at the index `i` can be the content of one of the node of the tree at the depth `i`.
    - It will always be `NULL` terminated, you can assume it.
- Your function must return `1` if all the element of `path` are present in the tree and if for an element of `path` at the index `i` is present in one of the node at the depth `i`. And of course, all the nodes must be linked to form a path.

If you want to test your function with the following example, you can download the following files:

- string_split - Prototype of the function: `char **string_split(const char *string, char separator);`
- free_str_array - Prototype of the function: `void free_str_array(char **array);`
- ntree_free - Prototype of the function: `void ntree_free(NTree *tree);`
- These files are object files. That means that they are compiled. You can link them with your `main` and other source files to form an executable.
- These files are compiled with the `-g` gcc flag.

Example:

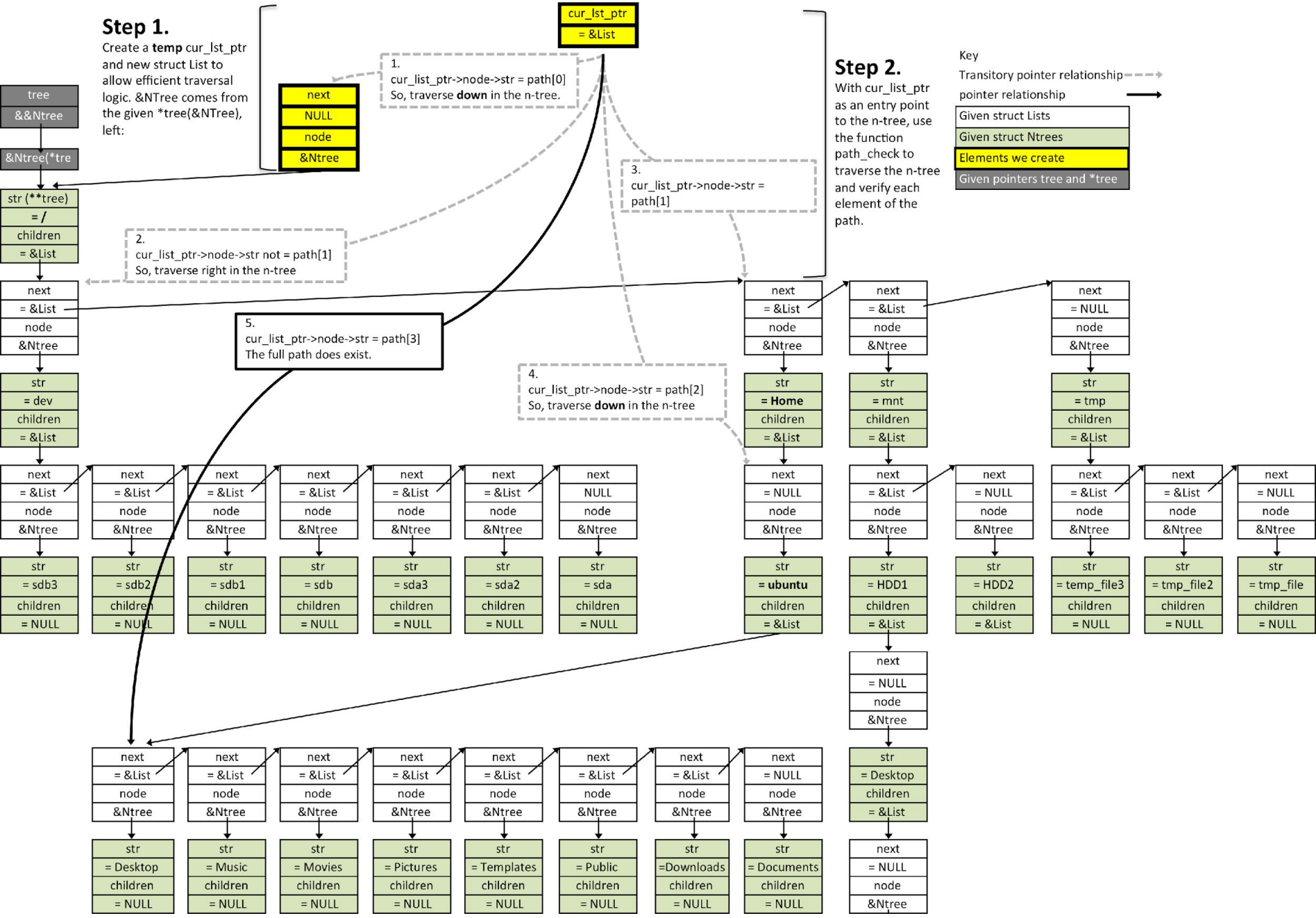
path_exists.c Task: Example main.c

```
alex@ubuntu:/tmp/ntrees$ cat main.c
#include <stdlib.h>
#include <stdio.h>
#include "tree.h"

int path_exists(NTree *, char **);

int ntree_insert(NTree **, char **, char *);
void ntree_free(NTree *);
char **string_split(const char *, char);
void free_str_array(char **);

int main(void)
{
    NTree *tree;
    char **array;

    tree = NULL;
    ntree_insert(&tree, NULL, "/");
    ntree_insert(&tree, (array = string_split("/", ',')), "home");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home", ' ')), "ubuntu");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu", ' ')), "Documents");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu", ' ')), "Download");
    free_str_array(array);
    ntree_insert(&tree, (array = string_split("/ home ubuntu", ' ')), "Public");
    free_str_array(array);

    printf("Path=[%s], present:%d\n", "/ home", path_exists(tree, (array = string_spli
t("/ home", ' '))));
    free_str_array(array);
    printf("Path=[%s], present:%d\n", "/ home ubuntu", path_exists(tree, (array = stri
ng_split("/ home ubuntu", ' '))));
    free_str_array(array);
    printf("Path=[%s], present:%d\n", "/ home ubuntu Download", path_exists(tree, (arr
ay = string_split("/ home ubuntu Download", ' '))));
    free_str_array(array);
    printf("Path=[%s], present:%d\n", "/ home student", path_exists(tree, (array = str
ing_split("/ home student", ' '))));
    free_str_array(array);
    printf("Path=[%s], present:%d\n", "/ home ubuntu Public file", path_exists(tree,
 (array = string_split("/ home ubuntu Public file", ' '))));
    free_str_array(array);

    ntree_free(tree);
    return (0);
}
```

path_exists.c Task: Expected Output

```
alex@ubuntu:/tmp/ntrees$ gcc -Wall -Wextra -Werror -pedantic -g main.c ntree_insert.c
 ntree_free.c path_exists.c string_split.c free_str_array.c
alex@ubuntu:/tmp/ntrees$
alex@ubuntu:/tmp/ntrees$
alex@ubuntu:/tmp/ntrees$ ./a.out
Path=[/ home], present:1
Path=[/ home ubuntu], present:1
Path=[/ home ubuntu Download], present:1
Path=[/ home student], present:0
Path=[/ home ubuntu Public file], present:0
alex@ubuntu:/tmp/ntrees$
```

Determine if a path exists using: path_exists(tree, (array = string_split("/ home ubuntu Desktop", ' '))));



Determine if a path exists using: path_exists(tree, (array = string_split("/ home ubuntu Desktop", ' '))));

Step 1.
Create a **temp** cur_lst_ptr

cur_lst_ptr
= &List

The basic logic:
Iterate through the path elements and traverse the tree.
If you find path[i] in the tree, go down the tree.
If you dont find path[i] in the tree, go right in the tree.
If at any point you run out of places to go, the path does not exist!

traverse the n-tree

Given pointers tree and *tree

```c
/**
 * path_check - Returns 1 if all elements of an array of char * are found in an
 * n-tree.
 *
 * @cur_lst_ptr: Pointer to the root struct List of an n-tree.
 * @path: An array of char * holding possible elements of an n-tree.
 *
 * Description: Given a pointer to a temporary struct List, serving as an entry
 * point to an n-tree, and an array of char *, the function traverses the n-tree
 * and determines if all elements of the char * array exist in the correct
 * hierarchy.
 */
int path_check(List *cur_lst_ptr, char **path)
{
    int array_len;
    int i;

    array_len = array_length(path);
    i = 0;
    while (i < array_len)
    {
        /* Check for a match in the current List/Ntree */
        if (strcmp(cur_lst_ptr->node->str, path[i]) == 0)
        {
            /*
             * If there is a match at the current List/NTree, and we haven't
             * yet reached the last element of path, check to see if there is
             * a lower List/NTree to hold subsequent elements of path. If false,
             * return 0 because there the full path cannot exist. If true, move
             * cur_lst_ptr to point to this lower List/NTree.
             */
            if (i < (array_len - 1))
            {
                /*
                 * If children holds NULL, the full path cannot exist because
                 * there is no lower List/NTree to hold subsequent path elements
                 * . return 0.
                 */
                if (cur_lst_ptr->node->children == NULL)
                {

                    return 0;
                }
                /* move cur_lst_ptr to point to this lower List/NTree */
                else
                {
                    cur_lst_ptr = cur_lst_ptr->node->children;
                }
            }
            /*
             * If there is a match at the current List/NTree, and we HAVE
             * reached the last element of path, the full path does exist.
             * return 1.
             */
            if (i == (array_len - 1))
            {
                return 1;
            }
            /*
             * All elements of path, except for the final element, have been
             * found. Iterate to next element of path and check it by repeating
             * the entire while loop.
             */
            i++;
        }
        else
        {
            /*
             * If there is no match in the current List/NTree and there is no
             * "next" (rightward) List/NTree to check, the path cannot exist
             * return 0.
             */
            if (cur_lst_ptr->next == NULL)
            {
                return 0;
            }
            else
            /*
             * If there is no match in the current List/NTree but there IS a
             * rightward List/NTree to check, move cur_lst_ptr to point to this
             * rightward List/NTree, and check it by repeating the entire while
             * loop.
             */
            {
                cur_lst_ptr = cur_lst_ptr->next;
            }
        }
    }
    return 1;
}
```

= &List | = &List | = &List | = &List | = &List | = &List

ntree_free.c Task

2. Free it mandatory

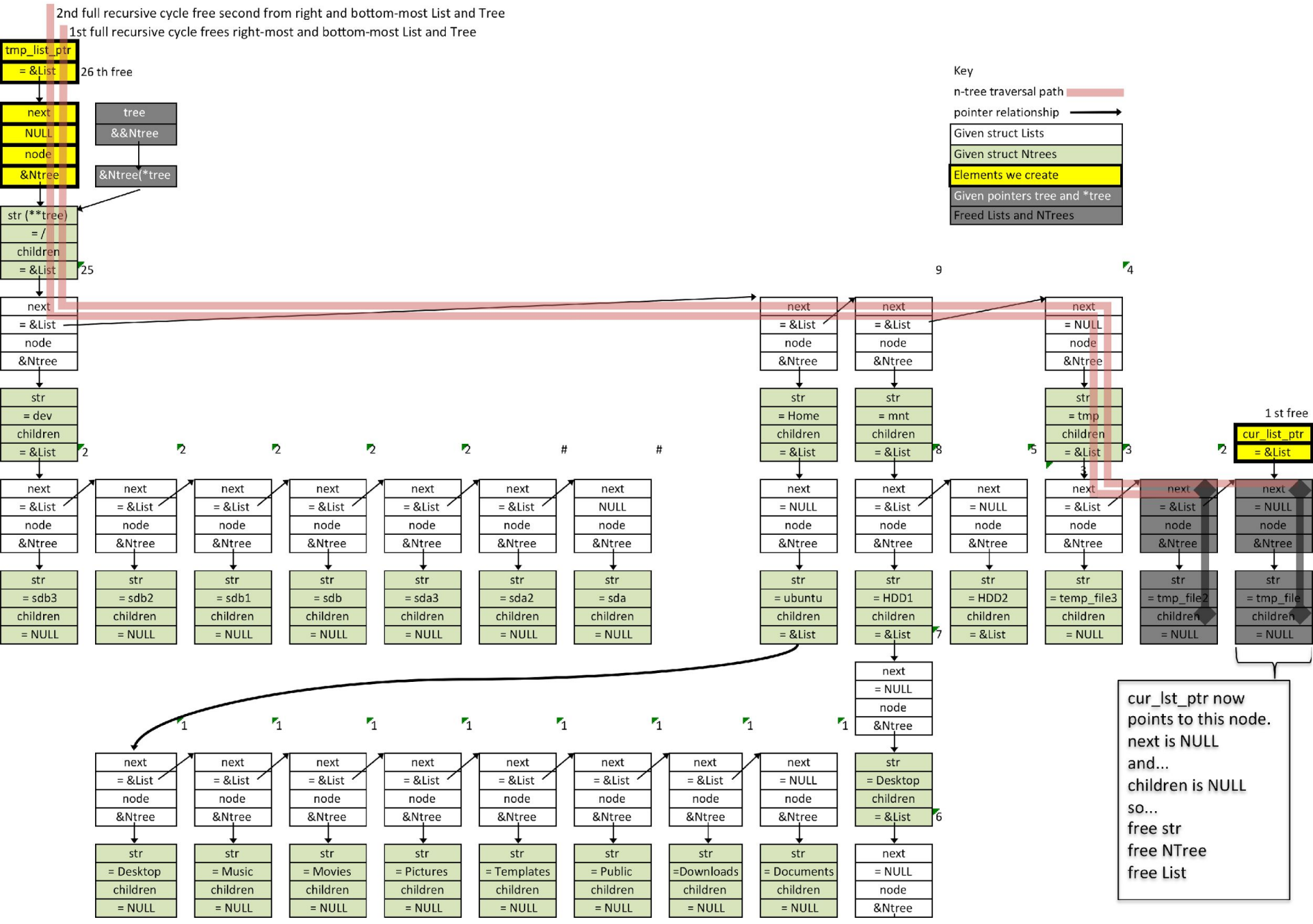
Write a function that free an entire N-ary tree

- Prototype: `void ntree_free(NTree *tree);`

Repo:

- GitHub repository: `holbertonschool-low_level_programming`
- Directory: `n_trees`
- File: `ntree_free.c`

Free an n-tree using the function tree_free(tree);



Free an n-tree using the function tree_free(tree);

2nd full recursive cycle free second from right and bottom-most List and Tree

1st full recursive cycle frees right-most and bottom-most List and Tree

The basic logic:
- **Whenever possible, recurse right**
- **Otherwise, recurse down**
- **When "next" is NULL and children is "NULL", you're at the bottom-right: free the str, NTree, and List**

```c
/**
 * free_list_ptr - Frees an n-tree.
 *
 * @cur_lst_ptr: Pointer to first List (TOP) of an n-tree
 *
 * Description: Given a temporary struct List and struct Ntree, the function
 * recursively traverses the ntree, and frees the List and NTree when both
 * next and children of the combined "super"struct List.node->children are NULL
 */
/* Free next, free children, and free cur lst ptr itself */
void free_list_ptr(List *cur_lst_ptr)
{
    if (cur_lst_ptr == NULL)
        return;
    if (cur_lst_ptr->next != NULL)
    {
        free_list_ptr (cur_lst_ptr->next);
    }
    if (cur_lst_ptr->node->children != NULL)
    {
        free_list_ptr (cur_lst_ptr->node->children);
    }
    free (cur_lst_ptr->node->str);
    free (cur_lst_ptr->node);
    free (cur_lst_ptr);
}
/**
 * ntree_free - Frees an n-tree.
 *
 * @tree: A pointer to the root struct of an NTree
 *
 * Description:  Given a pointer to the root struct of an NTree, the function
 * creates a temporary struct List and Struct NTree to complete a consistent
 * data structure, which then allows for a simple recursive traversal of the
 * tree in the called ftn free_list_ptr.
 */
void ntree_free(NTree *tree)
{
    List *cur_lst_ptr;
    /*create a list_ptr that points to root node*/
    cur_lst_ptr = new_list();
    cur_lst_ptr->next = NULL;
    cur_lst_ptr->node = tree;
    free_list_ptr(cur_lst_ptr);
}
```

Given struct Ntrees

Elements we create

Given pointers tree and *tree

Freed Lists and NTrees

cur_lst_ptr now points to this node.
next is NULL, and children is NULL, so...
free str, free NTree, free List