دانشکده علوم ریاضی و آمار

مدرس: دکتر مجتبی رفیعی **موضوع پروژه:** پیاده سازی درخت

ساختمان داده‌ها و الگوریتم‌ها
پروژه اختیاری

نام و نام خانوادگی: *محمد احمدی* ***مهلت تحویل:*** ۲۱ دی۱۴۰۳

- این پروژه اختیاری و حداقل یک نمره مازاد بر نمرات درس، به آن تعلق گرفته است.
- گروه‌بندی مجاز نیست و حل این تمرینات به صورت انفرادی انجام می‌شود.
- مهلت زمان ارسال پروژه تا بیست و یکم دی ماه است.
- برای هر پروژه یک متن در نظر گرفته شده است که هنگام تحویل سوال مشخص شده است.

ساختار پروژه

- دو فایل مربوط به LaTeX: یک فایل حاوی سورس و دیگری PDF مربوطه باشد.
  - توضیحات کلی درباره‌ی روش پیاده‌سازی در صورت نیاز:
    - شبه کد عملیات‌ها:
      - توضیح شبه‌کد؛
      - شبه‌کد.
    - تحلیل مجانبی از کارایی زمانی (و کارایی فضایی در صورت نیاز).
- فایل C:
  - تعریف دقیق و ساختارمند ساختار داده‌ها و اشیای مورد استفاده،
  - پیاده‌سازی عملیات‌های خواسته شده،
  - اجرای ورودی داده شده در صورت سوال،
  - کامنت‌گذاری در صورت نیاز.

پیاده سازی درخت

این پیاده‌سازی به طور کلی یک درخت پویا را تعریف می‌کند. درخت از گره‌هایی تشکیل شده که هر گره می‌تواند چندین فرزند داشته باشد. گره ها متشکل از بخش‌های زیر هستند.

۱. بخش داده: مقداری که گره نگهداری میکند.

۲. بخش والد: اشاره‌گری به والد گره. اگر گره ریشه باشد، مقدار این فیلد NULL است.

۳. بخش فرزندان: آرایه‌ای از اشاره‌گرها به فرزندان گره.

۴. بخش تعداد فرزندان گره.

شبه کدها

ایجاد گره:

Algorithm 1 CreateNode(data)

```
1: newNode.data ← data
2: newNode.parent ← NULL
3: newNode.children ← []
4: newNode.childCount ← 0
5:
6: return newNode
```

الگوریتم یک گره جدید ایجاد می‌کند و اطلاعات اولیه مربوط به آن گره را تنظیم می‌کند.
تحلیل کارایی زمانی: مجموع این عملیات‌ها ثابت هستند، زیرا هیچ‌یک از آن‌ها وابسته به اندازه داده‌ها یا پیچیدگی ساختار نیستند. بنابراین، پیچیدگی زمانی این الگوریتم از مرتبه $O(1)$ می‌باشد.

افزودن فرزند به گره :

Algorithm 2 AddChild(parent, child)

```
1: Resize parent.children to parent.childCount + 1
2: parent.children[parent.childCount] ← child
3: parent.childCount ← parent.childCount + 1
```

این الگوریتم مسئول افزودن یک گره جدید (فرزند) به یک گره والد در ساختار درختی است. فضای ذخیره‌سازی لیست children متعلق به parent افزایش داده می‌شود تا امکان افزودن یک فرزند جدید فراهم شود.
تحلیل کارایی زمانی: مجموع این عملیات‌ها ثابت هستند، بنابراین پیچیدگی زمانی این الگوریتم از مرتبه $O(1)$ می‌باشد.

محاسبه عمق گره:

Algorithm 3 NodeDepth(node)

```
depth ← 0
while node.parent ≠ NULL do
    depth ← depth + 1
    node ← node.parent
return depth
```

الگوریتم NodeDepth برای محاسبه عمق یک گره در یک ساختار درختی استفاده می‌شود. عمق یک گره به تعداد یال‌ها بین آن گره و ریشه درخت گفته می‌شود. در این الگوریتم، با حرکت به سمت والدین گره، عمق گره محاسبه می‌شود.
تحلیل کارایی زمانی: پیچیدگی زمانی این الگوریتم به تعداد گره‌های روی مسیر از گره فعلی تا ریشه بستگی دارد. اگر عمق گره d باشد تعداد تکرارهای حلقه d خواهد بود. در نتیجه پیچیدگی زمانی از مرتبه $O(d)$ میباشد.

بررسی رابطه پدر-فرزندی:

Algorithm 4 IsParentChild(parent, child)

```
if child.parent = parent then
    return True
else
    return False
```

الگوریتم IsParentChild برای بررسی این موضوع است که آیا یک گره خاص فرزند گره دیگری هست یا خیر.
تحلیل کارایی زمانی: این الگوریتم تنها شامل یک مقایسه ساده است و مستقل از اندازه درخت یا تعداد گره‌ها عمل می‌کند. بنابراین پیچیدگی زمانی آن $O(1)$ می‌باشد.

بررسی رابطه جد-فرزندی:

Algorithm 5 IsAncestor(ancestor, node)

```
while node ≠ NULL do
    if node.parent = ancestor then
        return True
    node ← node.parrent
return False
```

الگوریتم IsAncestor برای بررسی این که آیا یک گره خاص، جد یک گره دیگر هست یا خیر، استفاده می‌شود. این الگوریتم با حرکت از گره هدف به سمت ریشه درخت، والدین آن را بررسی می‌کند تا ببیند آیا یکی از آن‌ها برابر با ancestor هست.
تحلیل کارایی زمانی: پیچیدگی زمانی این الگوریتم به تعداد گره‌های روی مسیر از گره node تا ریشه بستگی دارد. اگر عمق گره d باشد، حلقه حداکثر d بار اجرا می‌شود. پس پیچیدگی از مرتبه $O(d)$ می‌باشد.

چاپ نوادگان یک گره:

Algorithm 6 NodeDescendants(node)

```
1: if node = NULL then
2:     return
3: for i ← 0 to node.childCount - 1 do
4:     Print node.children[i].data
5:     NodeDescendants(node.children[i])
```

الگوریتم NodeDescendants برای چاپ تمامی نوادگان (فرزندان، نوه‌ها، و به همین ترتیب) یک گره در ساختار درختی طراحی شده است. این الگوریتم از روش بازگشتی استفاده می‌کند و برای هر گره، تمام فرزندان آن و زیرشاخه‌های مربوطه را پیمایش و چاپ می‌کند.
تحلیل کارایی زمانی: برای تحلیل پیچیدگی زمانی، باید در نظر بگیریم که:

- هر گره دقیقاً یک بار بازدید می‌شود.
- در هر بازدید، فرزندان گره بررسی می‌شوند.

اگر تعداد کل گره‌های موجود در درخت را n در نظر بگیریم، کارایی زمانی از مرتبه $O(n)$ میباشد.

گرفتن والد یک گره:

Algorithm 7 GetParent(node)

```
1: return node.parent
```

این الگوریتم وظیفه دارد که والد یک گره خاص را برگرداند.
تحلیل کارایی زمانی: این الگوریتم تنها شامل یک عملیات ساده دسترسی به فیلد است، بنابراین پیچیدگی زمانی آن $O(1)$ می‌باشد.

گرفتن هم‌سطح‌ها:

Algorithm 8 GetSiblings(node)

```
1: if node.parent = NULL then
2:     return
3: parent ← node.parent
4: for i ← 0 to parent.childCount - 1 do
5:     if parent.children[i] ≠ node then
6:         Print parent.children[i].data
```

این الگوریتم برای پیدا کردن و چاپ داده‌های خواهر و برادرهای یک گره خاص در یک ساختار درختی استفاده می‌شود. خواهر و برادرها گره‌هایی هستند که دارای یک والد مشترک هستند اما خود گره مورد نظر (node) نیستند.
تحلیل کارایی زمانی: پیمایش تمام فرزندان والد: اگر تعداد فرزندان والد n باشد، حلقه باید n بار اجرا شود. در هر تکرار، یک مقایسه و چاپ انجام می‌شود، که هر کدام $O(1)$ زمان می‌برد. بنابراین، پیچیدگی زمانی این الگوریتم$O(n)$ می‌باشد.

۱ پیاده سازی با زبان C:

Listing 1: Tree implementation in C language

```c
#include <stdio.h>
#include <stdlib.h>

// define a structure template for a node in the tree.
typedef struct NodeStruct {
        int data;
        struct NodeStruct* parent;
        struct NodeStruct** children;
        int childCount;      // number of children the node currently has.
} NodeStruct;

NodeStruct* Create_Node(int data, NodeStruct* parent);
int Node_Depth(NodeStruct* node);
int Is_Parent_Child(NodeStruct* parent, NodeStruct* child);
int Is_Ancestor(NodeStruct* ancestor, NodeStruct* node);
int Is_Internal(NodeStruct* node);
void Node_Descendants(NodeStruct* node);
void Get_Siblings(NodeStruct* node);
void Add_Child(NodeStruct* parent, NodeStruct* child) ;   // DRY
void Free_Node(NodeStruct* node);

int main() {
        NodeStruct* root = Create_Node(1, NULL);
        NodeStruct* child1 = Create_Node(2, root);
        NodeStruct* child2 = Create_Node(3, root);
        Add_Child(root, child1);
        Add_Child(root, child2);

        printf("Depth of child1: %d\n", Node_Depth(child1));
        printf("Is root parent of child1? %d\n", Is_Parent_Child(root, child1));

        NodeStruct* grandchild = Create_Node(4, child1);
        Add_Child(child1, grandchild);

        printf("Is root ancestor of grandchild? %d\n", Is_Ancestor(root, grandchild));
        printf("Descendants of root: ");
        Node_Descendants(root);
        printf("\n");

        printf("Siblings of child1: ");
        Get_Siblings(child1);
        printf("\n");

        Free_Node(root);
        printf("memory freed successfully\n");
        return 0;
}

NodeStruct* Create_Node(int data, NodeStruct* parent) {
        // allocate memory for a new node and cast it to the appropriate type
        NodeStruct* newNode = (NodeStruct*)malloc(sizeof(NodeStruct));
        newNode->data = data;
        newNode->parent = parent;
        newNode->children = NULL;
        newNode->childCount = 0;
        return newNode;
}

int Node_Depth(NodeStruct* node) {
```

```
        int depth = 0;
        while (node->parent != NULL) {
                depth++;
                node = node->parent;
        }
        return depth;
}

int Is_Parent_Child(NodeStruct* parent, NodeStruct* child) {
        return (child->parent == parent);
}

int Is_Ancestor(NodeStruct* ancestor, NodeStruct* node) {
        while (node != NULL) {
                if (node->parent == ancestor) {
                        return 1;
                }
                node = node->parent;
        }
        return 0;
}

int Is_Internal(NodeStruct* node) {
        return (node->childCount > 0);
}

void Node_Descendants(NodeStruct* node) {
        if (node == NULL) return;
        for (int i = 0; i < node->childCount; i++) {
                printf("%d ", node->children[i]->data);
                Node_Descendants(node->children[i]);
        }
}

void Get_Siblings(NodeStruct* node) {
        if (node->parent == NULL) return;      // if node is root, return
        NodeStruct* parent = node->parent;
        for (int i = 0; i < parent->childCount; i++) {
                if (parent->children[i] != node) {
                        printf("%d ", parent->children[i]->data);
                }
        }
}

void Add_Child(NodeStruct* parent, NodeStruct* child) {
        // Resize the children array to accommodate the new child.
        parent->children = (NodeStruct**)realloc(parent->children, sizeof(NodeStruct*) * (
            parent->childCount + 1));
        parent->children[parent->childCount] = child;
        parent->childCount++;
}

// function to free node and its children recursively
void Free_Node(NodeStruct* node) {
        if (node == NULL) return;
        // free memory for all children
        for (int i = 0; i < node->childCount; i++) {
                Free_Node(node->children[i]);
        }
        // free children array
        free(node->children);

        free(node);
}
```