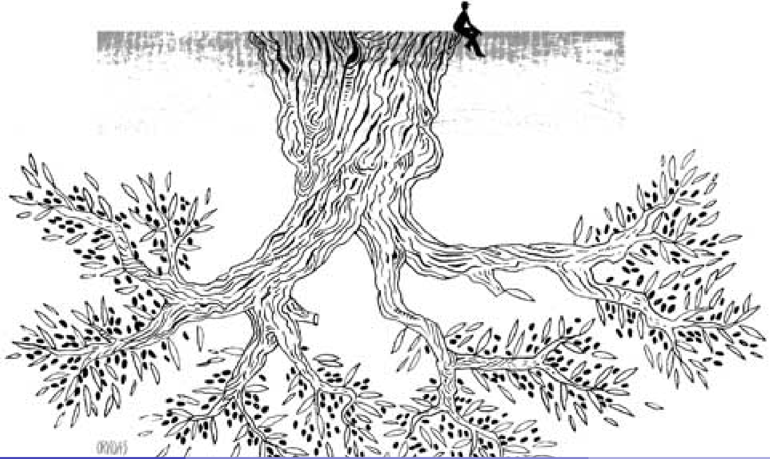
Cấu trúc dữ liệu và giải thuật
Chương 3 - Phần 1

Khoa CNTT, Đại học Kỹ thuật - Công nghệ Cần Thơ
Lưu hành nội bộ

Content

Khái niệm về cây - trees



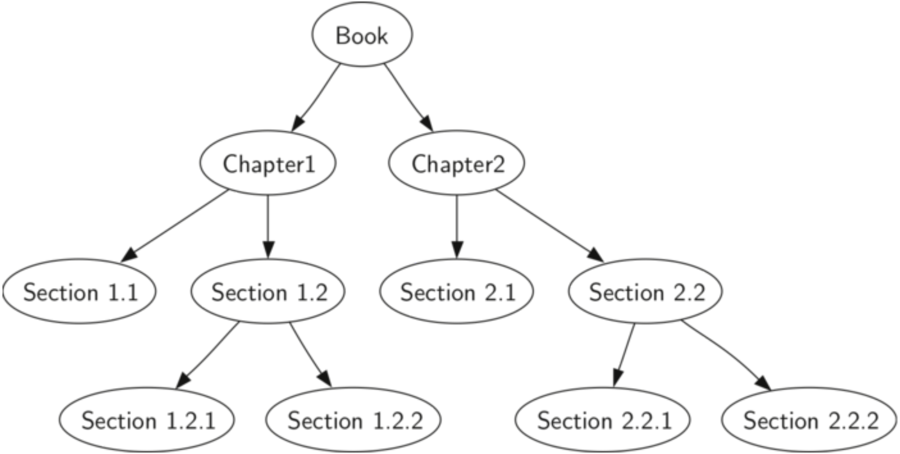
Khái niệm về cây - trees

- Cấu trúc dữ liệu phi tuyến tính quan trọng trong máy tính: cấu trúc cây.
- Cấu trúc cây phân chia dữ liệu theo từng nhóm có liên quan theo một định nghĩa cụ thể.
- Kiểu cấu trúc này cho phép giải thuật truy xuất dữ liệu nhanh hơn cấu trúc tuyến tính đối với các dữ liệu có tính phân nhóm cao.

Khái niệm về cây - trees

- Cấu trúc cây thể hiện tư duy tổ chức dữ liệu có thứ tự thứ bậc (hierarchical).
  - Cha - con; tổ tiên - hậu duệ; cây con.
- Các ví dụ sử dụng cấu trúc cây như cấu trúc tập tin và thư mục hệ thống, giao diện đồ họa người dùng, cơ sở dữ liệu, trang web.

Khái niệm về cây - trees



Định nghĩa hình thức

- Một cây `T` là một tập hợp các nút (`node`) chứa dữ liệu.
- Trong đó các nút có mối quan hệ cha-con `parent-child` và phải thỏa mãn một số điều kiện sau:
  - Nếu `T` không rỗng, `T` có một nút đặc biệt `r` không có cha, gọi là gốc của `T`.
  - Mỗi nút `u` khác `r` có một nút cha `v` duy nhất; các nút `u` có cùng cha `v` được gọi là các con của `v`.

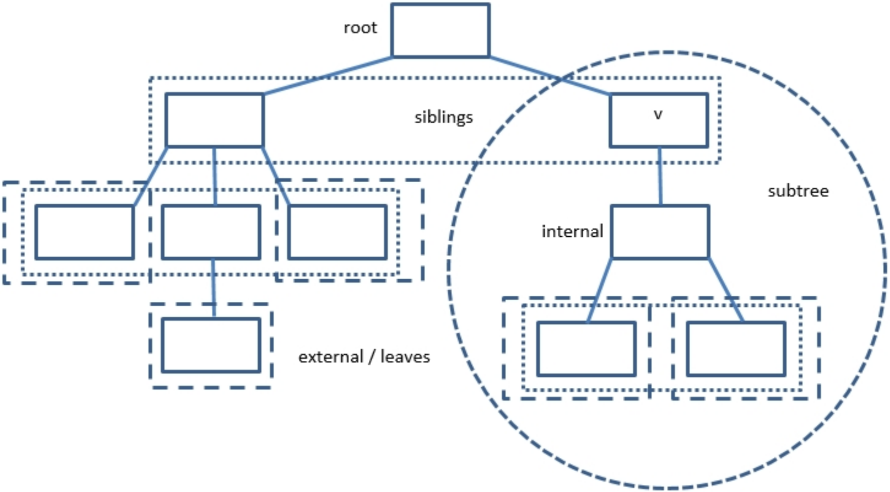
Mối quan hệ giữa các nút

- Hai hay nhiều nút có cùng cha được gọi là anh chị em (`siblings`).
- Một nút được gọi là nút ngoại (`external`) khi nút đó không có con nào.
- Nút ngoại còn được gọi là nút lá (`leaves`).
- Một nút được gọi là nút nội (`internal`) khi nút đó có một hoặc nhiều con.

Mối quan hệ giữa các nút

- Cây con (`sub-tree`) của cây `T` có gốc tại nút `v` bao gồm `v` và tất cả các nút con và các nút sau đó.
- Một đỉnh (`edge`) là một cặp nút (u, v) sao cho u và cha của v hoặc ngược lại.
- Một đường (`path`) là chuỗi các nút sao cho 2 nút liên tiếp trong chuỗi hình thành một đỉnh.

Mối quan hệ giữa các nút



1. Khái niệm về cây
2. Cài đặt cấu trúc dữ liệu trừu tượng cây
3. Cây nhị phân
4. Cài đặt cấu trúc dữ liệu trừu tượng cây nhị phân
5. Cài đặt cây nhị phân bằng cấu trúc liên kết
6. Các phương pháp duyệt cây

Cài đặt cây

- Đối với cấu trúc phi tuyến tính như cấu trúc cây, chúng ta định vị trí bằng `position`.
- Mỗi đối tượng được lưu trữ ở một `position` và các `position` thỏa mãn mối quan hệ cha-con.
- Một đối tượng `position` p hỗ trợ `method` sau: p`.element()` trả về giá trị lưu lại vị trí p.

Cài đặt cây

- Gọi `T` là một cấu trúc dữ liệu trừu tượng cây, ta có một số `method` sau:

  1. `T.root()`: trả về position của gốc `r` của `T`, hoặc trả về `None` nếu `T` rỗng.
  2. `T.is_root(p)`: trả về `True` nếu position `p` là gốc của `T`, ngược lại trả về `False`.
  3. `len(T)`: trả về số lượng position hiện có trong `T`.
  4. `T.is_empty()`: trả về `True` nếu `T` rỗng, ngược lại trả về `False`.

Cài đặt cây

5. `T.is_leaf(p)`: Trả về `True` nếu position p không có con nào, ngược lại trả về `False`.
6. `T.parent(p)`: trả về position của cha của position p, hoặc trả về `None` nếu position p là gốc của `T`.
7. `T.num_children(p)`: trả về số con của position p.
8. `T.children(p)`: trả về vòng lặp các con của position p, ngược lại trả về `None` nếu p là nút lá.

Cài đặt cây

9. `T.positions()`: trả về vòng lặp qua tất cả các position trong `T`, ngược lại trả về `none` nếu `T` rỗng.
10. `iter(T)`: trả về vòng lặp qua các dữ liệu lưu trong `T`, ngược lại trả về `none` nếu `T` rỗng.

Cài đặt cây

- Các method trên nhận vào một giá trị đúng `p`, ngược lại trả về lỗi `ValueError`.
- Nếu `T` là một cây có trật tự thì `T.children(p)` trả về các con của vị trí `p` theo mối quan hệ ý nghĩa.
- Ta sẽ xây dựng một `Tree class` trừu tượng bao gồm các method cơ bản được đã được mô tả.
- Sau đó, tùy các loại cây cụ thể, ta sẽ kế thừa `Tree class` trừu tượng này.

Cài đặt Tree class trừu tượng

- Tên file PyDev Module là `Tree.py`

```
class Tree(object):

    class Position(object):
        def element(self):
            ''' return the element stored at a position'''
            raise NotImplementedError("must be implemented by subclass")

        def __eq__(self, other):
            '''return T if p stays at the same position'''
            raise NotImplementedError("must be implemented by subclass")

        def __ne__(self, other):
            return not (self == other)
```

```python

    def root(self):
        ''' return the root of a tree. Must be implemented by subclass '''
        raise NotImplementedError("must be implemented by subclass")

    def is_root(self, p):
        ''' return True if p is the root, otherwise return False '''
        return self.root() == p

    def is_leaf(self, p):
        ''' return True if p is a leaf, otherwise return False '''
        return self.num_children(p) == 0

    def is_empty(self):
        ''' return True if the tree is empty '''
        return len(self) == 0
```

```python

    def __len__(self):
        ''' return the total number of positions in the tree. Must be implemeted by subclass '''
        raise NotImplementedError("must be implemented by subclass")

    def parent(self, p):
        ''' return the parent of position p, or return None if p is root. Must be implemeted by subclass '''
        raise NotImplementedError("must be implemented by subclass")

    def children(self, p):
        ''' return the children of position p, or return None if p is leaf. Must be implemeted by subclass '''
        raise NotImplementedError("must be implemented by subclass")
```

```python

    def num_children(self):
        ''' return the number of children that the position p has. Must be implemented by subclass '''
        raise NotImplementedError("must be implemented by subclass")

    def depth(self, p):
        ''' return the number of ancestors of p '''
        if self.is_root(p):
            return 0
        else:
            return 1 + self.depth(self.parent(p))
```

```python

    def _height_recursive(self, p):
        ''' return the max height of position p '''
        if self.is_leaf(p):
            return 0
        else:
            return 1 + max(self._height_recursive(c) for c in self.children(p))

    def height(self, p=None):
        ''' return the max height of the subtree at position p. If p is None, return the height of the entire tree.
        It is a convenient way to calculate the height of the entire tree instead of defining a specific method '''
        if p is None:
            p = self.root()
        return self._height_recursive(p)
```

Chiều sâu của cây

- Gọi `p` là một position của một nút trong cây `T`.
- Chiều sau của position `p` là khoảng cách từ `p` đến gốc của cây, không tính bản thân `p`.
- Hàm `def depth(self, p)` có ý nghĩa như sau:
  - Nếu `p` là gốc, thì chiều sâu của `p` là 0.
  - Ngược lại, chiều sâu của `p` là 1+ chiều sâu của cha của `p`.
  - Phương pháp gọi hàm trong hàm là phương pháp đệ quy (recursion).

Chiều cao của cây

- Gọi `p` là một position của một nút trong cây `T`.
- Chiều cao của position `p` là **khoảng cách lớn nhất** từ `p` đến một lá, không tính bản thân `p`.
- Hàm `def _height_recursive(self, p)` có ý nghĩa như sau:
  - Nếu `p` là lá thì chiều cao của `p` là 0.
  - Ngược lại, chiều cao của `p` là $1+$ chiều cao lớn nhất của các con của `p`.

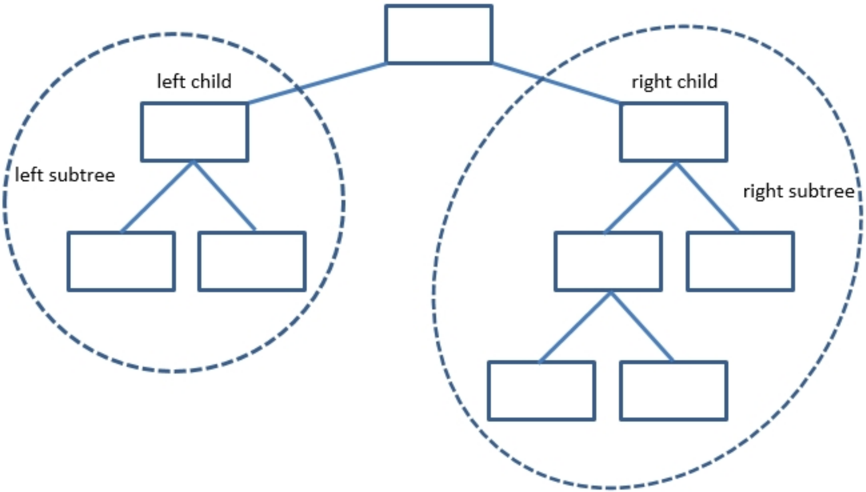
1. Khái niệm về cây
2. Cài đặt cấu trúc dữ liệu trừu tượng cây
3. Cây nhị phân
4. Cài đặt cấu trúc dữ liệu trừu tượng cây nhị phân
5. Cài đặt cây nhị phân bằng cấu trúc liên kết
6. Các phương pháp duyệt cây

Định nghĩa cây nhị phân

- Một cây nhị phân (`binary tree`) là một cây có trật tự (ordered tree) có các tính chất sau:
    - Mỗi nút có nhiều nhất 2 con.
    - Mỗi một con của nút chỉ được gán một trong hai nhãn: con trái (`left child`) và con phải (`right child`).
    - Thứ tự con trái đứng trước con phải.

Định nghĩa cây nhị phân

- Cây nhị phân được gọi là phù hợp (`proper`) nếu mỗi nút hoặc có 0 hoặc có 2 con (proper binary tree).
- Cây nhị phân được gọi là hoàn toàn (`full`) nếu mỗi nút nội có chính xác 2 con (full binary tree).

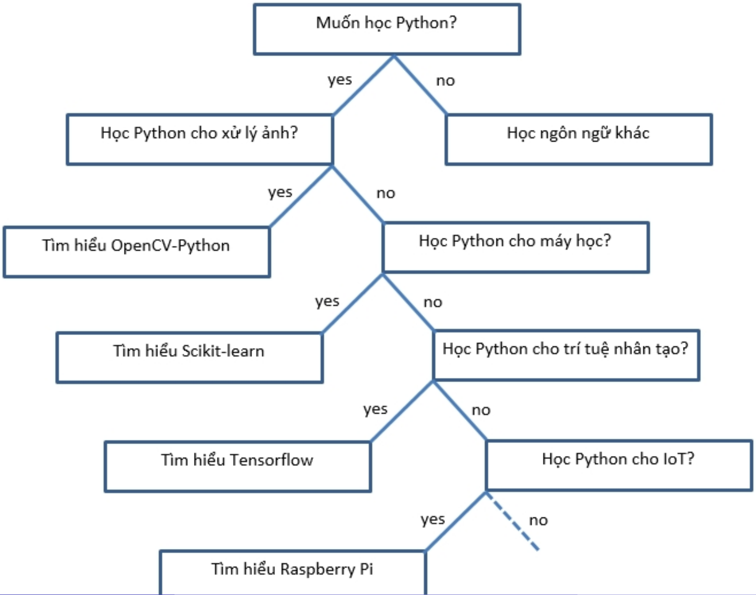

left child
right child
left subtree
right subtree

Ngữ cảnh sử dụng

- Mô hình hóa bài toán được thể hiện bằng một chuỗi các câu hỏi đúng sai (yes-no).
- Cây nhị phân được biết đến như cây quyết định (decision trees).
    - Mỗi một nút nội tương tứng với một câu hỏi.
    - Bắt đầu từ gốc, chúng ta duyệt về bên trái hay phải tùy vào câu trả lời đúng hoặc sai.
    - Với mỗi quyết định, chúng ta lần lượt duyệt đỉnh từ nút cha đến nút con, tuần từ từ đỉnh đến nút lá.

Ngữ cảnh sử dụng

- Chú ý, không có quy tắc bắt buộc bên trái, bên phải là `đúng`, `sai` hay ngược lại.
- Tuy nhiên, nên nhất quán về vị trí của `đúng` và `sai` cho toàn bộ cây.


Muốn học Python?
yes
no
Học Python cho xử lý ảnh?
Học ngôn ngữ khác
yes
no
Tìm hiểu OpenCV-Python
Học Python cho máy học?
yes
no
Tìm hiểu Scikit-learn
Học Python cho trí tuệ nhân tạo?
yes
no
Tìm hiểu Tensorflow
Học Python cho IoT?
yes
no
Tìm hiểu Raspberry Pi

Cài đặt cây nhị phân

- Cấu trúc dữ liệu trừu tượng cây nhị phân là trường hợp cụ thể của cấu trúc dữ liệu trừu tượng cây.
- Cấu trúc dữ liệu trừu tượng cây nhị phân sẽ kế thừa các hàm phát triển cho cây nói chung và phát triển riêng cho tính chất của nó.

Kế thừa

- Kế thừa class `Tree`, cây nhị phân sẽ mặc nhiên sử dụng các hàm đã được phát triển, ví dụ `parent`, `is_leaf()`.
- kế thừa class `Position` vì nó được cài đặt lòng ghép vào bên trong của lớp `Tree`.

Cài đặt cây nhị phân

- Gọi `T` là một cấu trúc dữ liệu trừu tượng cây nhị phân, p là một vị trí (position) trong cây, ta có một số `method` sau:
    - `T.left(p)`: trả về con trái của p, hoặc trả về `None` nếu p không có con trái.
    - `T.right(p)`: trả về con phải của p, hoặc trả về `None` nếu p không có con phải.
    - `T.sibling(p)`: trả về anh em của p, hoặc trả về `None` nếu p không có anh em.
    - `T.children(p)`: trả về các con của p, hoặc trả về rỗng nếu p là nút lá.

Cài đặt class BinaryTree trừu tượng

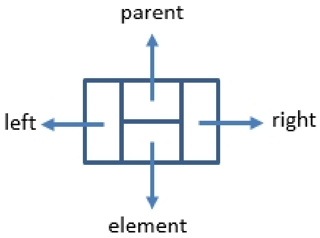
- File PyDev Module được lưu là BinaryTree.py.

```
from Tree import Tree

class BinaryTree(Tree):
    ''' Abstract definition of a binary tree '''

    def left(self, p):
        ''' return the p's left child position. Return None if p has no child '''
        raise NotImplementedError("must be implemented by subclass")

    def right(self, p):
        ''' return the p's right child position. Return None if p has no right child '''
        raise NotImplementedError("must be implemented by subclass")
```

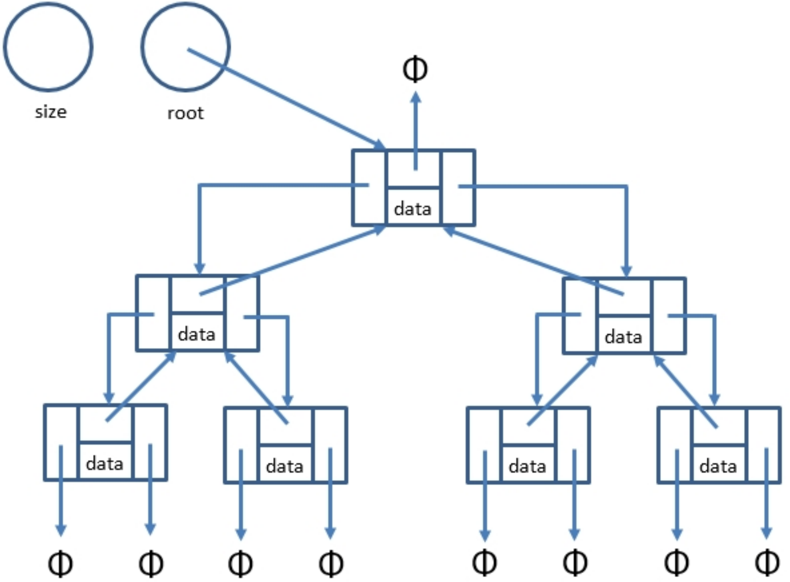
```python

    def siblings(self, p):
        ''' return the p's sibling. Return None if there is no sibling
     '''
        parent = self.parent(p)
        if parent is None:  # check if p is root
            return None
        else:
            if p == self.left(parent):
                return self.right(parent)
            else:
                return self.left(parent)

    def children(self, p):
        ''' return an iteration of p's children '''
        if self.left(p) is not None:
            yield self.left(p)
        if self.right(p) is not None:
            yield self.right(p)
```

Cấu trúc liên kết

- Cấu trúc liên kết (`linked structure`).
- Một số tài liệu gọi còn gọi là cấu trúc con trỏ vì mỗi nút trong cây có nhiều con trỏ.



- Ngoài ra bản thân cây nhị phân có biến trỏ đến gốc và biến lưu kích thước hay số lượng nút đang có.


size
root
Φ
data
data
data
data
data
data
data
Φ
Φ
Φ
Φ
Φ
Φ
Φ
Φ

Cấu trúc liên kết

- Đối với cây nhị phân liên kết, ta định nghĩa thêm một số hàm dành riêng cho đối tượng:
  1. `T.add_root(e)`: tạo một nút gốc cho 1 cây rỗng, lưu trữ *e* như dữ liệu tại nút, trả về vị trí của gốc. Lỗi xảy ra nếu cây không rỗng.
  2. `T.add_left(p, e)`: tạo một nút mới, lưu trữ *e* như dữ liệu tại nút, liên kết nút mới là con trái của p, trả về vị trí của nút. Lỗi xảy ra nếu p đã có con trái.
  3. `T.add_right(p, e)`: tạo một nút mới, lưu trữ *e* như dữ liệu tại nút, liên kết nút mới là con phải của p, trả về vị trí của nút. Lỗi xảy ra nếu p đã có con phải.

4. `T.replace(p, e)`: Thay thế dữ liệu đang lưu tại p bằng giá trị e, trả về dữ liệu trước khi thay thế.

5. `T.delete(p)`: Xóa nút tại vị trí p, thay thế p bằng con của nó, trả về dữ liệu lưu tại p. Lỗi xảy ra nếu p có hai con.

6. `T.attach(p, T1, T2)`: gắn cây T1 và T2 vào p là cây con trái và cây con phải tương ứng, thiết lập cây T1 và T2 là rỗng. Lỗi xảy ra nếu p không phải nút lá.

Cài đặt class LinkedBinaryTree

- File PyDev Module được lưu là `LinkedBinaryTree.py`.

```
from BinaryTree import BinaryTree

class LinkedBinaryTree(BinaryTree):

    class _Node(object):
        ''' Create a nested Node class'''
        def __init__(self, element, parent = None, left = None, right = None):
            self._element = element
            self._parent = parent
            self._left = left
            self._right = right
```

```
    class Position(BinaryTree.Position):
        def __init__(self, container, node):
            self._container = container
            self._node = node

        def element(self):
            ''' return the data stored '''
            return self._node._element

        def __eq__(self, other):
            ''' return True if current position and other position is the
    same location '''
            return type(other) is type(self) and other._node is self._node
```

```python
    def _make_position(self, node):
        ''' return position of node. Return None if there is no node '''
        return self.Position(self, node) if node is not None else None

    def __init__(self):
        ''' Constructor for a concrete object '''
        self._root = None
        self._size = 0

    def __len__(self):
        ''' return the total number of nodes in the tree '''
        return self._size

    def root(self):
        ''' return the root position of the tree '''
        return self._make_position(self._root)
```

```
    def parent(self, p):
        ''' return the p's parent position '''
        node = p._node
        return self._make_position(node._parent)

    def left(self, p):
        ''' return the p's left child position. Return None if there is no child '''
        node = p._node
        return self._make_position(node._left)

    def right(self, p):
        ''' return the p's right child position. Return None if there is no right child '''
        node = p._node
        return self._make_position(node._right)
```

```python
    def num_children(self, p):
        ''' return the number of children of p '''
        node = p._node
        count = 0
        if node._left is not None:
            count += 1
        if node._right is not None:
            count += 1
        return count
```

```
def _add_root(self, e):
    ''' add element e at the root of an empty tree. return its
 position. ValueError if the tree is nonempty '''
    if self._root is not None:
        raise ValueError("The tree is nonempty")
    self._size += 1
    self._root = self._Node(e)
    return self._make_position(self._root)
```

```
    def _add_left(self, p , e):
        ''' create a new left child of p, store element e, return new node's position. Raise error if left child of p already exists '''
        node = p._node
        if node._left is not None:
            raise ValueError("left child exists")
        self._size += 1
        node._left = self._Node(e, node)
        return self._make_position(node._left)
```

```
    def _add_right(self, p , e):
        ''' create a new right child of p, store element e, return new node's position. Raise error if right child of p already exists '''
        node = p._node
        if node._right is not None:
            raise ValueError("right child exists")
        self._size += 1
        node._right = self._Node(e, node)
        return self._make_position(node._right)

    def _replace(self, p, e):
        ''' replace current element by new e, return the old one '''
        node = p._node
        old = node._element
        node._element = e
        return old
```

```
    def _delete(self, p):
        ''' delete the node at p, replace it with its child, if any.
        return the element stored at p.
        Raise error if p is invalid or p has two children. '''
        node = p._node
        if self.num_children(p) == 2:
            raise ValueError("p has two children")
        child = node._left if node._left else node._right
        if child is not None:
            child._parent = node._parent
        if node is self._root:
            self._root = child
        else:
```

```
            parent = node._parent
            if node is parent._left:
                parent._left = child
            else:
                parent._right = child
        self._size -= 1
        node._parent = node
        return node._element
```

Kiểm thử cài đặt

- Ta viết 1 chương trình nhỏ kiểm thử cài đặt vừa tạo.
- Tên file `LinkedBinaryTree_Example.py`.

```
from LinkedBinaryTree import LinkedBinaryTree

if __name__ == '__main__':
    lbt = LinkedBinaryTree()
    print(lbt.root())
    print(lbt.is_empty())
    print(len(lbt))
    print("="*30)

    root = lbt._add_root("A")
    print(lbt.is_empty())
    print(len(lbt))
    print(root._node._element)
```

```python
    print(root._node._parent)
    print(root._node._left)
    print(root._node._right)
    print(lbt.num_children(root))
    print("="*30)

    b = lbt._add_left(root, "B")
    c = lbt._add_right(root, "C")
    print(len(lbt))
    print(lbt.height(root))
    print(lbt.depth(b))
    print(lbt.num_children(root))
    print(b._node._element)
    print(b._node._parent)
    print(b._node._parent._element)
    print(b._node._left)
    print(b._node._right)
    print("="*30)
```

```
    d = lbt._add_left(b, "D")
    e = lbt._add_right(b, "E")
    print(len(lbt))
    print(lbt.height(root))
    print(lbt.depth(d))
    print(e._node._element)
    print(e._node._parent)
    print(e._node._parent._element)
    print(e._node._left)
    print(e._node._right)
    print("="*30)

    print(lbt.num_children(b))
    print(b._node._left._element)
    print(b._node._right._element)
    print("="*30)
```

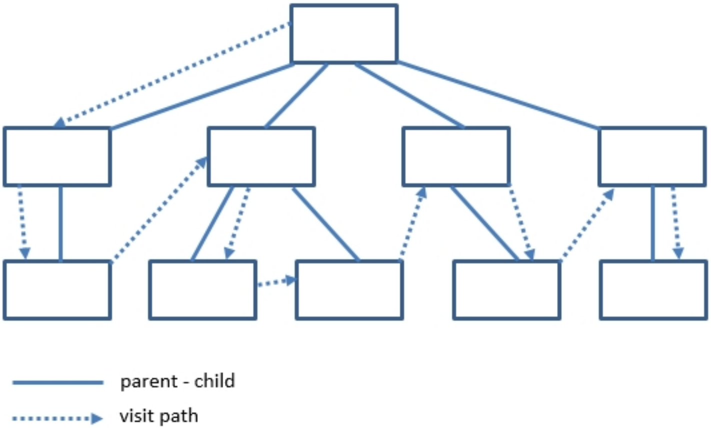
```
    lbt._delete(d)
    print(lbt.num_children(b))
    print((lbt.left(b)))
    print(lbt.right(b))
    print(lbt.right(b)._node._element)
    print("="*30)
```

Độ phức tạp thuật toán

Tác vụ	Thời gian thực thi
`len`, `is_empty`	$\mathcal{O}(1)$
`root`, `parent`, `left`, `right`, `sibling`, `children`, `num_children`	$\mathcal{O}(1)$
`is_root`, `is_leaf`	$\mathcal{O}(1)$
`depth`	$\mathcal{O}(d_p + 1)$
`height`	$\mathcal{O}(n)$
`add_root`, `add_left`, `add_right`, `replace`, `delete`	$\mathcal{O}(1)$

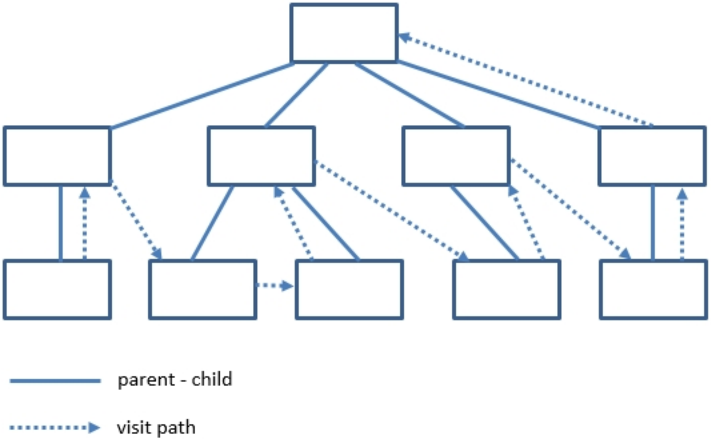
Duyệt thứ bậc trước

- Duyệt thứ bậc trước (`preorder traversal`) của một cây `T`, gốc của `T` được duyệt trước và đến lần lượt các cây con một cách đệ quy.
- Nếu cây có trật tự thì duyệt cây con tương ứng với duyệt theo thứ tự các con.


parent - child
visit path

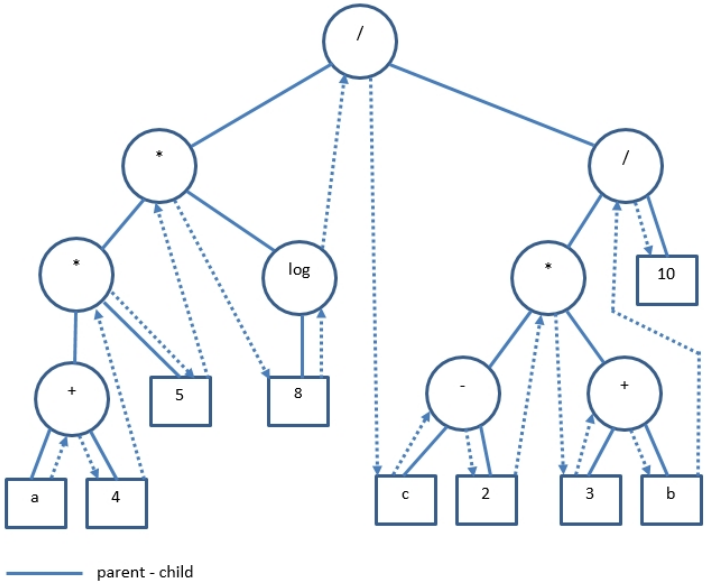
Duyệt thứ bậc sau

- Duyệt thứ bậc sau (`postorder traversal`) đối nghịch với duyệt trước ở thứ tự duyệt các nút.
- Các con của nút p được duyệt trước và sau cùng sẽ duyệt nút p.


parent - child
visit path

Duyệt thứ bậc ngang cây nhị phân

- Hai phương pháp duyệt thứ bậc trước và duyệt thứ bậc sau đối với cấu trúc cây đều có thể được áp dụng hoàn toàn cho cây nhị phân.
- Ta có một phương pháp duyệt riêng đối với cây nhị phân đó là duyệt thứ bậc ngang (`inorder traversal`).
- Tại mỗi vị trí cần duyệt `p`, thức tự duyệt sẽ là con trái của `p`, bản thân `p` và con phải của `p`.


/
*
/
*
log
*
10
+
5
8
-
+
a
4
c
2
3
b
parent - child