Chuyên Đề Tổ Chức Dữ Liệu
Bài Tập 3

Trần Minh Đức
20810206

1. Splay tree

Sinh viên tìm hiểu một dạng cây nhị phân tìm kiếm là splay tree (https://en.wikipedia.org/wiki/Splay_tree). Trình bày các đặc điểm chính, cài đặt các thao tác và viết chương trình minh họa việc dùng cây này.

A. Đặc điểm:

Splay Tree là một loại cây nhị phân tìm kiếm tự cân bằng (self-adjusting binary search tree). Điểm đặc biệt của nó là mỗi khi bạn truy cập (thêm, xóa hoặc tìm) một nút, cây sẽ tự đưa nút đó lên gốc bằng một chuỗi các phép xoay (rotation). Mục tiêu là giữ cho các nút được truy cập thường xuyên gần gốc → truy cập nhanh hơn.

B. Cấu trúc cơ bản:

Mỗi nút có: key, left, right (và đôi khi có parent).
Tuân theo quy tắc của Binary Search Tree (BST):

- left.key < node.key < right.key.

Splay Tree có ba thao tác chính:

- Splay (đưa nút lên gốc)
- Insert (thêm nút mới)
- Delete (xóa nút)
- Hai thao tác Insert và Delete đều gọi Splay để đảm bảo cân bằng.

C. Cơ chế Splay (quan trọng nhất)

Khi truy cập vào một nút (tìm, thêm, hoặc sau khi xóa):

- Dùng các phép xoay để đưa nút đó lên gốc.

Có 3 dạng xoay chính:

- **Zig**: Khi nút cần splay là con trực tiếp của gốc. → Chỉ cần xoay 1 lần (giống xoay đơn BST).
- **Zig-Zig**: Khi nút và cha nó cùng hướng (đều là con trái hoặc đều là con phải). → Thực hiện 2 lần xoay cùng hướng:
    - Xoay cha lên trước
    - Rồi xoay nút lên
- **Zig-Zag**: Khi nút và cha nó khác hướng (một trái, một phải). → Thực hiện 2 xoay ngược hướng:
    - Xoay nút lên cha
    - Rồi xoay tiếp nút lên ông

D. Thao tác thêm (Insert)

- Thêm nút như trong BST bình thường.
- Sau khi thêm xong → splay nút mới lên gốc.

E. Thao tác xóa (Delete)

Splay nút cần xóa lên gốc.
Sau khi ở gốc:

- Nếu không có cây con trái → gốc = cây con phải.
- Nếu có cây con trái:
    - Lấy cây con trái ra.
    - Tìm nút phải nhất trong cây con trái (nút lớn nhất bên trái).
    - Splay nút đó lên gốc của cây con trái.
    - Gán cây con phải (của gốc ban đầu) làm cây con phải của nút đó.

F. Mục tiêu và đặc tính

Không đảm bảo chiều cao cân bằng tuyệt đối.
Nhưng trung bình (amortized): mọi thao tác có độ phức tạp O(log n).
Các phần tử thường xuyên truy cập sẽ nằm gần gốc.

G. Ưu và nhược điểm

Ưu điểm:

- Đơn giản hơn AVL hay Red-Black Tree (không lưu thêm thông tin cân bằng).
- Tự điều chỉnh theo tần suất truy cập.

Nhược điểm:

- Hiệu năng có thể tệ nếu truy cập dữ liệu ngẫu nhiên.
- Không tối ưu cho mọi thao tác như cây cân bằng cứng (AVL, RBT).

H. Code minh hoạ:

Định nghĩa struct:

```
struct Node {
 int key;
 Node *left, *right;
 Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

struct SplayTree {
 Node *root;
 SplayTree() : root(nullptr) {}
 void insert(int key);
 void remove(int key);
 Node* search(int key);
 void print();
};
```

Các function xoay:

```
// Right rotate
Node *rightRotate(Node *x) {
 Node *y = x->left;
 x->left = y->right;
 y->right = x;
 return y;
}

// Left rotate
Node *leftRotate(Node *x) {
 Node *y = x->right;
 x->right = y->left;
 y->left = x;
 return y;
}
```

Function splay:

```
Node *splayUtil(Node *root, int key) {
 // Base cases: root is nullptr or key is at root
 if (root == nullptr || root->key == key)
   return root;

 // Key lies in left subtree
 if (root->key > key) {
   // Key is not in tree, we are done
   if (root->left == nullptr)
     return root;

   // Zig-Zig (Left Left)
   if (root->left->key > key) {
     // First recursively bring the key as root of left-left
     root->left->left = splayUtil(root->left->left, key);
     // Do first rotation for root
     root = rightRotate(root);
   }
   // Zig-Zag (Left Right)
   else if (root->left->key < key) {
     // First recursively bring the key as root of left-right
     root->left->right = splayUtil(root->left->right, key);
     // Do first rotation for root->left
     if (root->left->right != nullptr)
       root->left = leftRotate(root->left);
   }

   // Do second rotation for root
   return (root->left == nullptr) ? root : rightRotate(root);
 }
 // Key lies in right subtree
 else {
   // Key is not in tree, we are done
   if (root->right == nullptr)
     return root;

   // Zig-Zag (Right Left)
```

```
    if (root->right->key > key) {
      // Bring the key as root of right-left
      root->right->left = splayUtil(root->right->left, key);
      // Do first rotation for root->right
      if (root->right->left != nullptr)
        root->right = rightRotate(root->right);
    }
    // Zig-Zig (Right Right)
    else if (root->right->key < key) {
      // Bring the key as root of right-right
      root->right->right = splayUtil(root->right->right, key);
      // Do first rotation for root
      root = leftRotate(root);
    }

    // Do second rotation for root
    return (root->right == nullptr) ? root : leftRotate(root);
  }
}
```

Function Insert:

```
// Helper function to insert a key (BST insert)
Node *insertBST(Node *node, int key) {
 // If tree is empty, return new node
 if (node == nullptr)
   return new Node(key);

 // Otherwise, recur down the tree
 if (key < node->key)
   node->left = insertBST(node->left, key);
 else if (key > node->key)
   node->right = insertBST(node->right, key);
 // If key already exists, don't insert

 return node;
}

// Insert a key: insert like normal BST, then splay to root
void SplayTree::insert(int key) {
 root = insertBST(root, key);
 root = splayUtil(root, key);
}
```

Function search:

```
Node* SplayTree::search(int key) {
 root = splayUtil(root, key);
 return root->key == key ? root : nullptr;
}
```

Function remove:

```
void SplayTree::remove(int key) {
 root = splayUtil(root, key);
 if (root->key == key) {
   Node *left = root->left;
   Node *right = root->right;
   delete root;
   if (!left) {
     root = right;
   } else {
     // find the largest node in the left subtree
     Node *largest = left;
     while (largest->right) {
       largest = largest->right;
     }
     root = splayUtil(left, largest->key);
     root->right = right;
   }
 }
}
```

Function sprint:

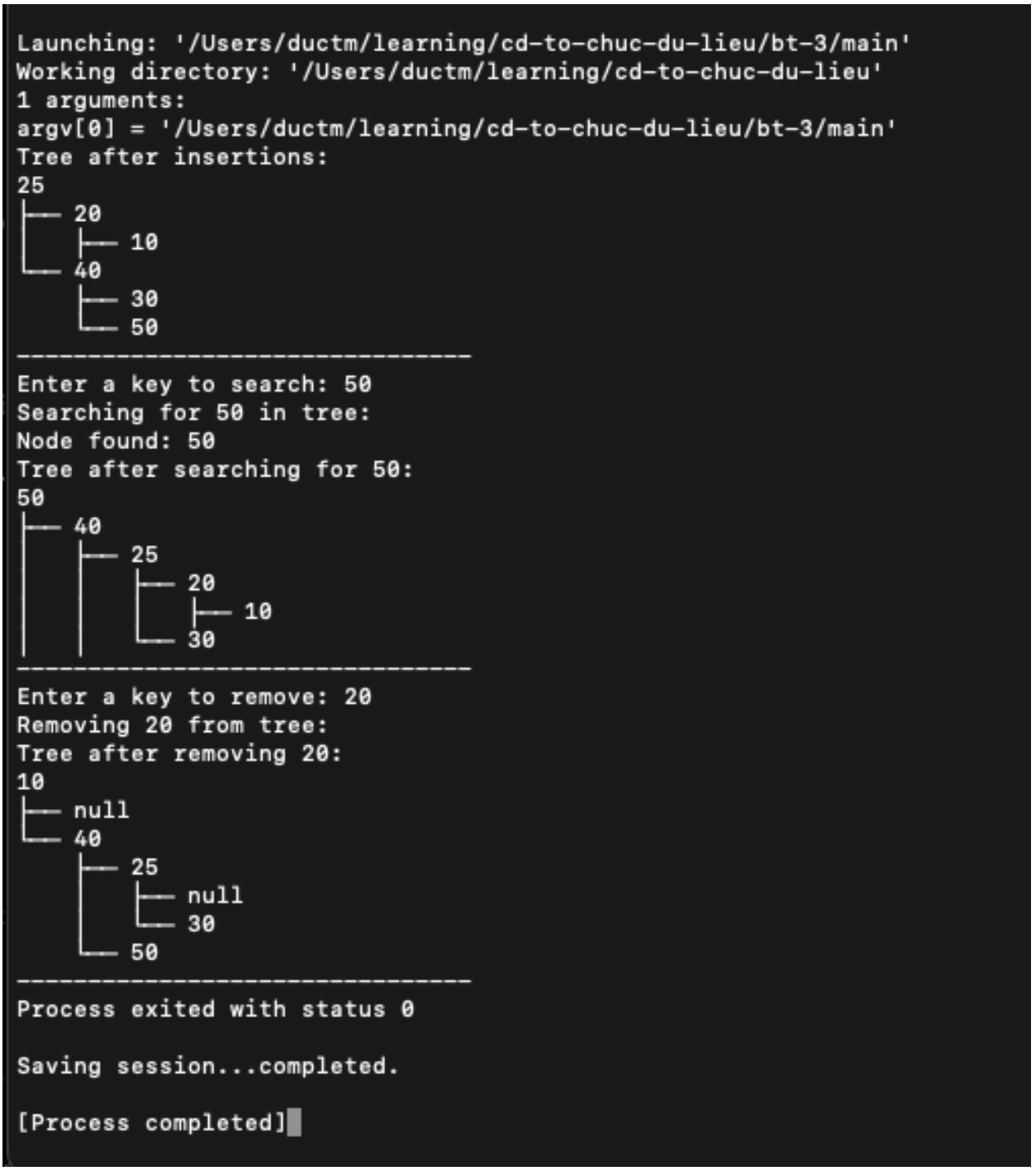
```
void printTree(Node *root, string prefix = "", bool isLeft =
true) {
 if (!root)
   return;

 cout << prefix;
 cout << (isLeft ? "├── " : "└── ");
 cout << root->key << endl;

 if (root->left || root->right) {
   if (root->left) {
     printTree(root->left, prefix + (isLeft ? "│   " : "
"), true);
   } else if (root->right) {
     cout << prefix << (isLeft ? "│   " : "    ") << "├── "
```

```
          << "null" << endl;
    }

    if (root->right) {
      printTree(root->right, prefix + (isLeft ? "│   " : "
"), false);
    }
  }
}

void SplayTree::print() {
 if (!root) {
   cout << "Empty tree" << endl;
   return;
 }
 cout << root->key << endl;
 if (root->left) {
   printTree(root->left, "", true);
 } else if (root->right) {
   cout << "└── null" << endl;
 }
 if (root->right) {
   printTree(root->right, "", false);
 }
 cout << "---------------------------------" << endl;
};
```

Function main để chạy ví dụ:

```
int main() {
 SplayTree tree;

 tree.insert(10);
 tree.insert(20);
 tree.insert(30);
 tree.insert(40);
 tree.insert(50);
 tree.insert(25);
 cout << "Tree after insertions: " << endl;
 tree.print();

 int input;
 cout << "Enter a key to search: ";
 cin >> input;
 cout << "Searching for " << input << " in tree: " << endl;
 Node* result = tree.search(input);
 if (result) {
   cout << "Node found: " << result->key << endl;
 } else {
   cout << "Node not found" << endl;
 }
 cout << "Tree after searching for " << input << ": " << endl;
 tree.print();

 cout << "Enter a key to remove: ";
 cin >> input;
 cout << "Removing " << input << " from tree: " << endl;
 tree.remove(input);
 cout << "Tree after removing " << input << ": " << endl;
 tree.print();


 return 0;
}
```

Kết quả sau khi chạy:



2. Mineven

Giả sử cấu trúc một nút của cây nhị phân chứa số nguyên được khai báo như sau:

```
struct NODE {
        int data;
        NODE *left, *right;
};
```

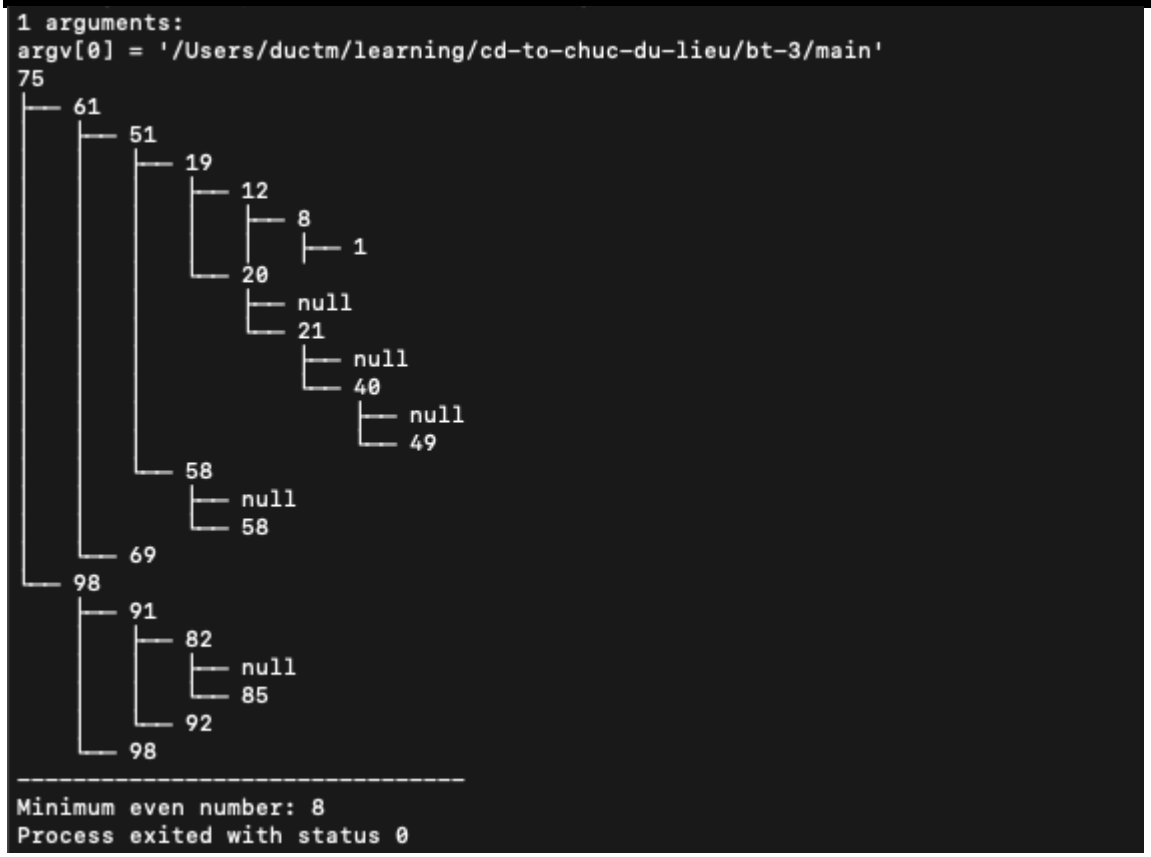
Hãy viết hàm NODE* mineven(NODE *root) để tìm nút mang phần tử chẵn nhỏ nhất của cây nhị phân có nút gốc trỏ bởi root bằng kĩ thuật đệ qui.

Hàm minieven

```
Node* mineven(Node *root) {
 if (!root) {
   return nullptr;
 }
 // tìm node chẵn nhỏ nhất bên trái
 Node *left = mineven(root->left);
 if (left) {
   return left;
 }
 // nếu node đang xét là node chẵn thì trả về node đó
 if (root->data % 2 == 0) {
   return root;
 }
 // nếu node đang xét không phải là node chẵn thì tiếp tục
tìm kiếm bên phải
 return mineven(root->right);
}
```

Kết quả chạy thử trên cây có 20 node ngẫu nhiên

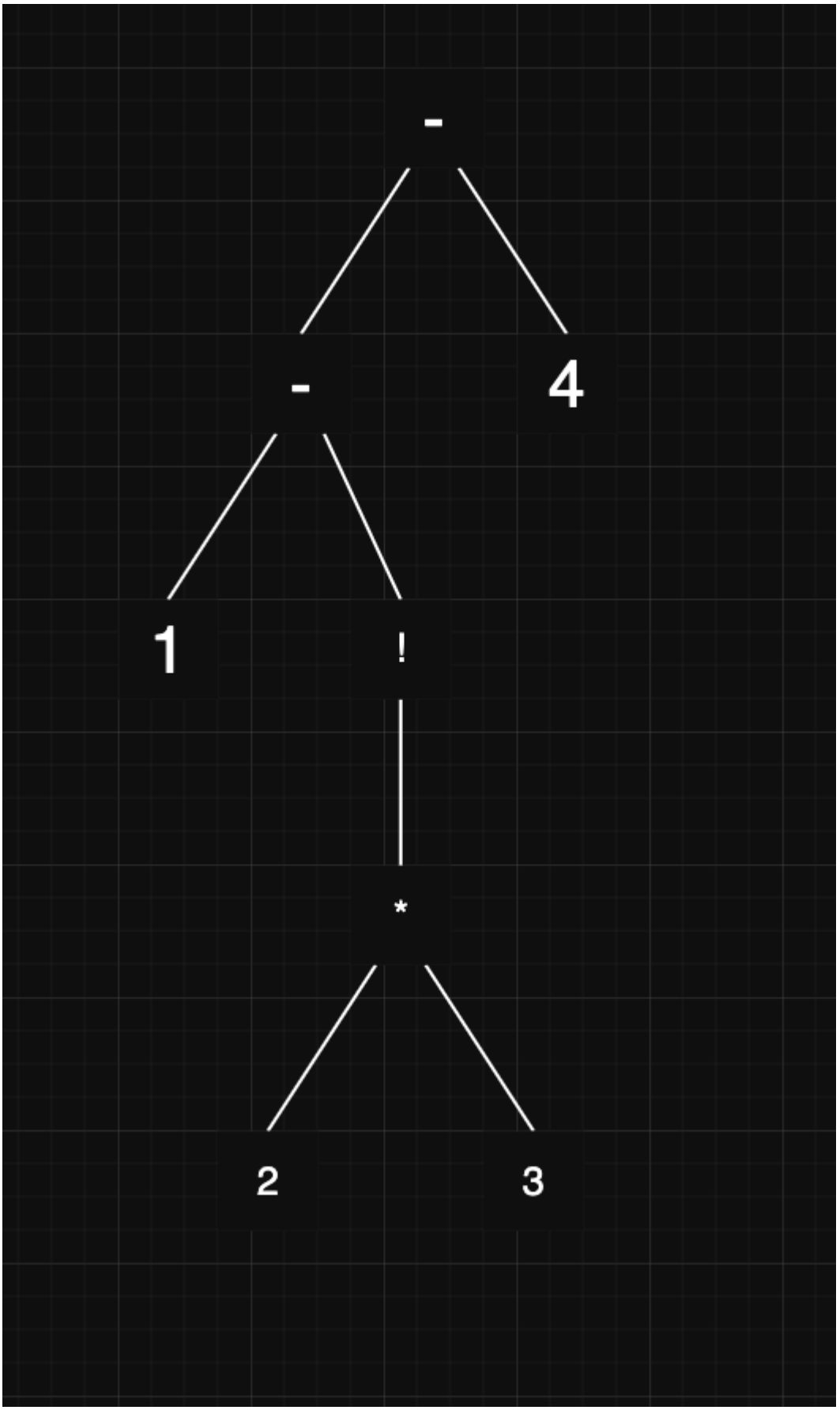
```
int main() {
 BinaryTree tree;
 // insert 10 random numbers
 srand(time(nullptr));
 for (int i = 0; i < 20; i++) {
   tree.insert(rand() % 100);
 }
 tree.print();
 Node *result = mineven(tree.root);
 if (result) {
   cout << "Minimum even number: " << result->data << endl;
 } else {
   cout << "No even number found" << endl;
 }
 return 0;
}
```



3. Cây biểu thức số học

a) Vẽ cây biểu thức số học của biểu thức 1 - (2 * 3)! - 4
b) Duyệt trước cây (a) để in ra biểu thức dạng tiền tố
c) Duyệt sau cây (a) để in ra biểu thức dạng hậu tố

Vẽ cây



Duyệt trước cây

=> - - 1 ! * 2 3 4

Duyệt sau cây

=> 1 2 3 * ! - 4 -