CSE221 Data Structures
Lecture 14: Binary Trees

Antoine Vigneron

antoine@unist.ac.kr

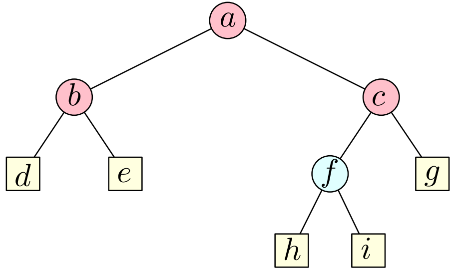
Ulsan National Institute of Science and Technology

November 3, 2021

Introduction

- I updated attendance records. They can be found in the portal under E-attendance.
- I will grade the midterm this week.
- Assignment 2 was graded by Hyeyun Yang (`gm1225@unist.ac.kr`.
- I will post Assignment 3 by the end of the week.
- Reference for this lecture: Textbook Section 7.3.

Binary Trees



b is the left child of a

c is the right child of a

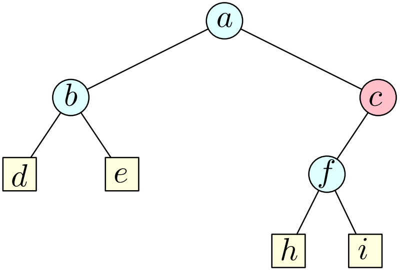
Definition

A *binary tree* is an ordered tree in which:

1. Every node has at most two children.
2. Each child node is labeled as being either a left child or a right child.
3. A left child precedes a right child in the ordering of children of a node.

Binary Trees

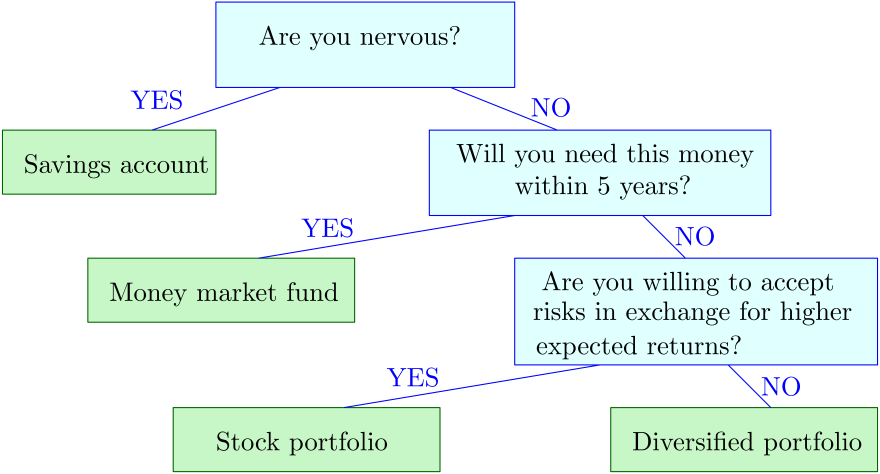
- The binary tree in the previous slide is a *full* binary tree: Each node has either 0 or 2 children.
- Some binary trees are not full:



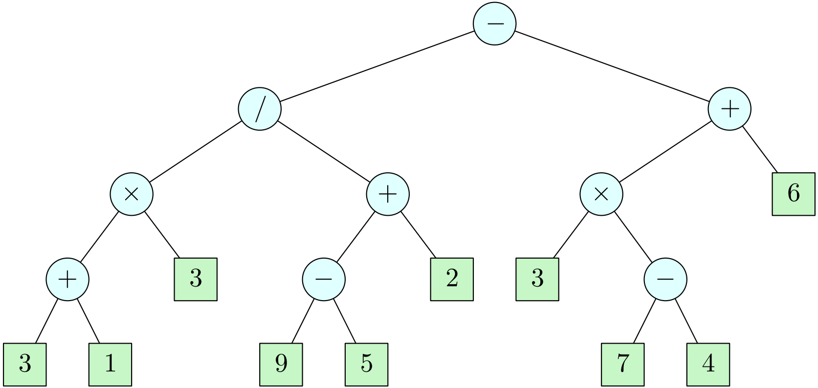
c has only one child

- All the binary trees in this course will be full, unless specified otherwise. Most applications involve full binary trees.

Example: A Decision Tree



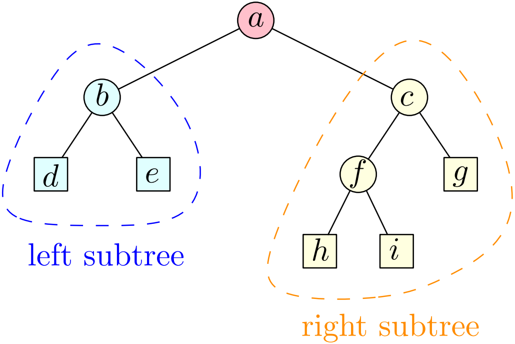
Example: Arithmetic-Expression Tree



Arithmetic-expression tree representing the arithmetic expression

$$(((3+1)\times 3)/((9-5)+2)) + (3\times(7-4)+6)$$

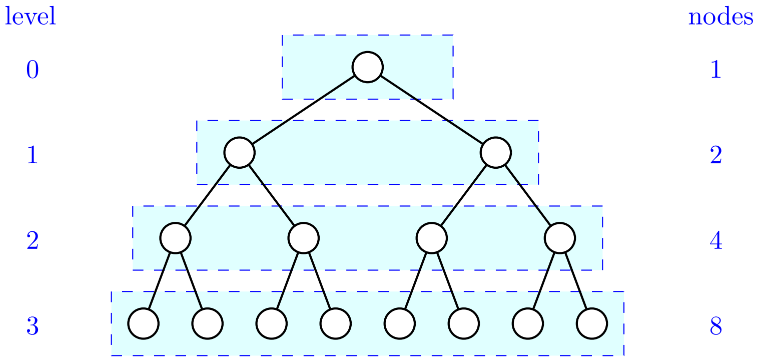
Recursive Definition



Definition

A binary tree is either empty, or consists of a root connected to two binary trees called the *left subtree* and the *right subtree*.

Properties



- Level i of a binary tree consists of at most 2^i nodes.

Properties

Proposition

Let T be a non-empty, full binary tree with n nodes, among which n_L are leaves and n_I are internal nodes. Let h be the height of T.

1. $2h + 1 \leqslant n \leqslant 2^{h+1} - 1$
2. $h + 1 \leqslant n_L \leqslant 2^h$
3. $h \leqslant n_I \leqslant 2^h - 1$
4. $\log(n + 1) - 1 \leqslant h \leqslant (n - 1)/2$
5. $n_L = n_I + 1$

- Sketch of proof given in class.

The Binary Tree ADT

- As with our general tree ADT, each node is associated with a position object p. The element stored at this node is given by $*p$. It supports the following operations:
- p.**left**(): Return the left child of p; an error condition occurs if p is a leaf.
- p.**right**(): Return the right child of p; an error condition occurs if p is a leaf.
- p.**parent**(): Return the parent of p; an error occurs if p is the root.
- p.**isRoot**(): Return true if p is the root and false otherwise.
- p.**isLeaf**(): Return true if p is a leaf and false otherwise.

The Binary Tree ADT

- The tree T itself supports the following operations:
- T.**size**(): Return the number of nodes in the tree.
- T.**empty**(): Return true if the tree is empty and false otherwise.
- T.**root**(): Return a position for the tree's root; an error occurs if the tree is empty.
- T.**positions**(): Return a position list of all the nodes of the tree.

C++ interface

C++ Code

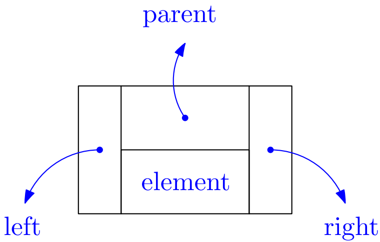
```
template <typename E>                         // base element type
class Position<E> {                           // a node position
public:
  E& operator*();                             // get element
  Position left() const;                      // get left child
  Position right() const;                     // get right child
  Position parent() const;                    // get parent
  bool isRoot() const;                        // root of tree?
  bool isLeaf() const;                        // a leaf?
};
```

C++ interface

C++ Code

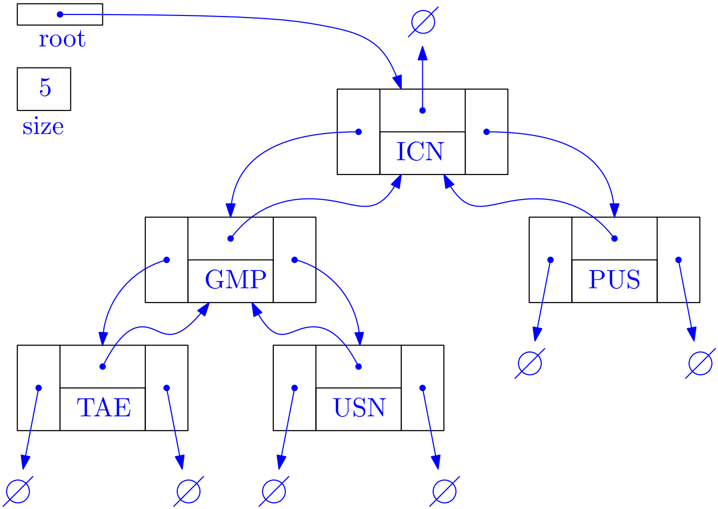
```
template <typename E>                          // base element type
class BinaryTree<E> {                                // binary tree
public:                                             // public types
  class Position;                                 // a node position
  class PositionList;                         // a list of positions
public:                                         // member functions
  int size() const;                               // number of nodes
  bool empty() const;                              // is tree empty?
  Position root() const;                              // get the root
  PositionList positions() const;                    // list of nodes
};
```

Linked Structure



- A node in the linked structure for full binary trees.

Linked Structure



Linked Structure

C++ Code

```
struct Node {                                    // a node of the tree
  Elem elt;                                          // element value
  Node* par;                                               // parent
  Node* left;                                          // left child
  Node* right;                                        // right child
  Node() : elt(), par(NULL), left(NULL), right(NULL) { }
                                                      // constructor
};
```

- Remark: This `struct` has a constructor. This is possible because a `struct` in C++ is essentially the same as a class, except that all its members are public by default.

```
class Position {                          // position in the tree
private:
  Node* v; // pointer to the node
public:
  Position(Node* _v = NULL) : v(_v) { }     // constructor
  Elem& operator*() { return v->elt; }         // get elt
  Position left() const { return Position(v->left); }
  Position right() const { return Position(v->right); }
  Position parent() const  { return Position(v->par); }
  bool isRoot() const { return v->par == NULL; }
  bool isLeaf() const                              // a leaf?
    { return v->left == NULL && v->right == NULL; }
  friend class LinkedBinaryTree;       // give tree access
};
typedef std::list<Position> PositionList;
                                       // list of positions
```

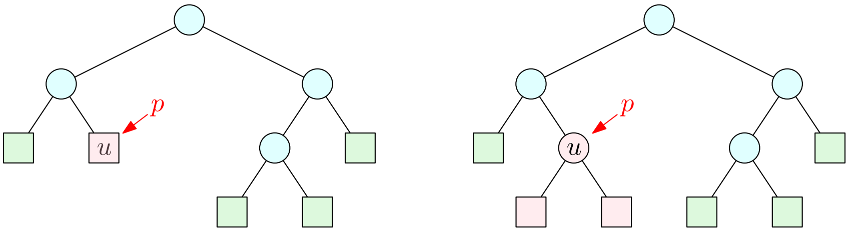
Linked Structure

```
typedef int Elem;                              // base element type
class LinkedBinaryTree {
protected:
  // insert Node declaration here ...
public:
  // insert Position declaration here ...
public:
  LinkedBinaryTree();                               // constructor
  int size() const;                             // number of nodes
  bool empty() const;                            // is tree empty?
  Position root() const;                           // get the root
  PositionList positions() const;                 // list of nodes
  void addRoot();                         // add root to empty tree
  void expandLeaf(const Position& p);             // expand leaf
  Position removeAboveLeaf(const Position& p);
  // housekeeping functions omitted. . .
```

```
protected:                                  // local utilities
  void preorder(Node* v, PositionList& pl) const;
                                           // preorder utility
private:
  Node* _root;                          // pointer to the root
  int n;                                    // number of nodes
};
```

```
LinkedBinaryTree::LinkedBinaryTree()           // constructor
  : _root(NULL), n(0) { }
int LinkedBinaryTree::size() const        // number of nodes
  { return n; }
bool LinkedBinaryTree::empty() const
  { return size() == 0; }                      // is tree empty?
LinkedBinaryTree::Position LinkedBinaryTree::root()
  { return Position( _root); } const           // get the root
void LinkedBinaryTree::addRoot() // add root to empty tree
  { _root = new Node; n = 1; }
```

Linked Structure



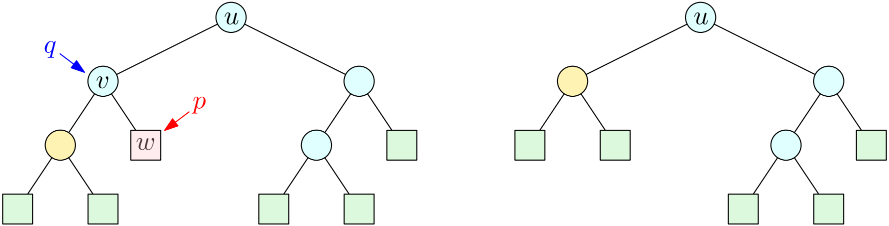
- **expandLeaf**(p): Transform p from a leaf into an internal node by creating two leaves and making them the left and right children of p, respectively; an error condition occurs if p is an internal node.

Linked Structure

C++ Code

```
                                                    // expand leaf
void LinkedBinaryTree::expandLeaf(const Position& p) {
  Node* v = p.v;                                    // p's node
  v->left = new Node;                   // add a new left child
  v->left->par = v;                          // v is its parent
  v->right = new Node;                // and a new right child
  v->right->par = v;                         // v is its parent
  n += 2;                                     // two more nodes
}
```

Linked Structure



- **removeAboveLeaf(*p*)**: Remove the leaf *p* together with its parent *q*, replacing *q* with the sibling of *p*; an error condition occurs if *p* is an internal node or *p* is the root.

```
LinkedBinaryTree::Position                    // remove p and parent
LinkedBinaryTree::removeAboveLeaf(const Position& p) {
  Node* w = p.v; Node* v = w->par;
  Node* sib = (w == v->left ? v->right : v->left);
  if (v == _root) {                                 // child of root?
    _root = sib;                                    // make sibling root
    sib->par = NULL;
  }
  else {
    Node* gpar = v->par;                            // w's grandparent
    if (v == gpar->left) gpar->left = sib;
    else gpar->right = sib;                    // replace parent by sib
    sib->par = gpar;
  }
  delete w; delete v;                          // delete removed nodes
  n -= 2;                                          // two fewer nodes
  return Position(sib);
}
```

Linked Structure

```
LinkedBinaryTree::PositionList             // list of all nodes
    LinkedBinaryTree::positions() const {
  PositionList pl;
  preorder( root, pl);                     // preorder traversal
  return PositionList(pl);                 // return resulting list
}
```

```
void LinkedBinaryTree::preorder            // preorder traversal
    (Node* v, PositionList& pl) const {
  pl.push_back(Position(v));               // preorder traversal
  if (v->left != NULL)                     // traverse left subtree
    preorder(v->left, pl);
  if (v->right != NULL)                    // traverse right subtree
    preorder(v->right, pl);
}
```

Analysis

Operation	time
left, right, parent, isLeaf, isRoot	$O(1)$
size, empty	$O(1)$
root	$O(1)$
expandLeaf, removeAboveLeaf	$O(1)$
positions	$O(n)$

Traversal

Pseudocode

procedure $\textsc{binaryPreorder}(T, p)$
  perform the "visit" action for node p
  if p is an internal node **then**
    $\textsc{binaryPreorder}(T, p.\text{left})$
    $\textsc{binaryPreorder}(T, p.\text{right})$

procedure $\textsc{binaryPostorder}(T, p)$
  if p is an internal node **then**
    $\textsc{binaryPostorder}(T, p.\text{left})$
    $\textsc{binaryPostorder}(T, p.\text{right})$
  perform the "visit" action for node p

Application

- Postorder traversal can be used to evaluate an arithmetic expression stored in a binary tree.
- For instance, on the tree pictured in Slide 7, the result should be 17.

Pseudocode

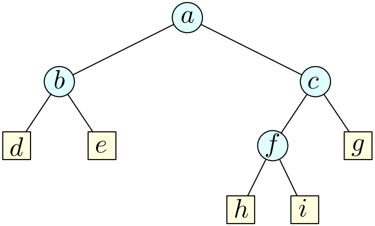
procedure $\textsc{Evaluate}(T, p)$
    if p is an internal node **then**
        $x \leftarrow \textsc{Evaluate}(T, p.\text{left})$
        $y \leftarrow \textsc{Evaluate}(T, p.\text{right})$
        Let $\circ$ be the operator stored at p
        return $x \circ y$
    return the value stored at p

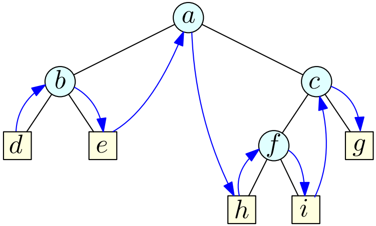
Inorder Traversal

Pseudocode

procedure INORDER(T, p)
    if p is an internal node **then**
        INORDER(T, p.left)
    perform the "visit" action for node p
    if p is an internal node **then**
        INORDER(T, p.right)

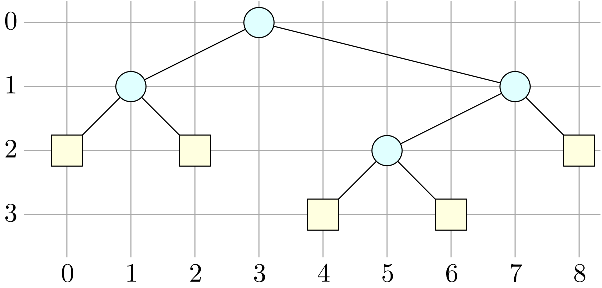
Inorder Traversal





- Output: d, b, e, a, h, f, i, c, g
- So we are visiting the nodes from left to right.

Inorder Traversal



- Application: Inorder traversal can be used for drawing a tree.
- The x-coordinate is given by the position in the inorder traversal.
- The y-coordinate is the depth.