CS121 Data Structures A, C
Binary Trees

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Binary Trees

A **binary tree** is an *ordered tree* with the following properties:

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

Recursive definition: a binary tree is either empty or consists of a root node together with left and right subtrees, both of which are binary trees

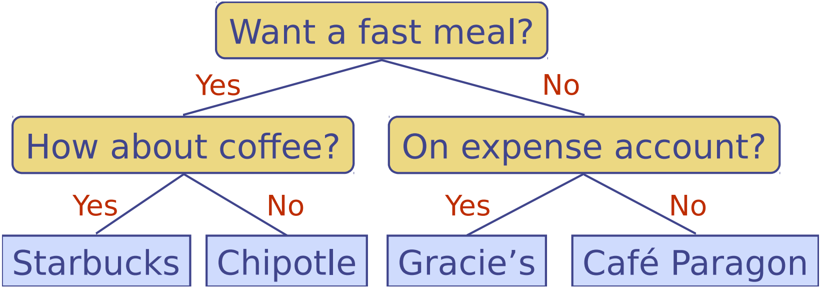
A binary tree is **proper**, or **full**, if each node has either zero or two children. Otherwise, it is **improper**.

The subtree rooted at a left or right child of an internal node v is called a **left subtree** or **right subtree**, respectively, of v

Decision Trees

Binary tree associated with a decision process

- internal nodes: questions with yes/no answer
- external nodes: decisions

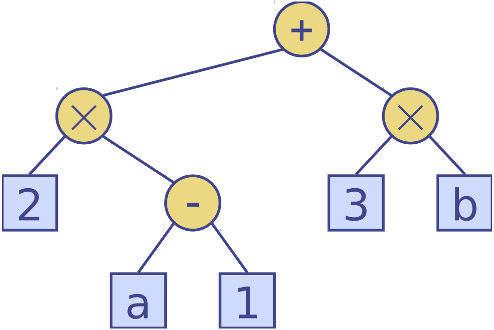


Is a decision tree proper or improper?

Arithmetic Expression Tree

Binary tree associated with an arithmetic expression

- internal nodes: operators
- external nodes: operands



What expression does this tree correspond to?

How can we determine the value associated with each node?

The Binary Tree ADT

The binary tree ADT is a specialisation of a tree with three additional **accessor** methods:

- left(p): Returns the position of the left child of p (or `null` if p has no left child)
- right(p): Returns the position of the right child of p (or `null` if p has no right child)
- sibling(p): Returns the position of the sibling of p (or `null` if p has no sibling)

Possible update methods will be considered when we discuss specific implementations and applications of binary trees

A BinaryTree Interface in Java

```
/** An interface for a binary tree, in which
    each node has at most two children. */
public interface BinaryTree<E> extends Tree<E> {
  /** Returns the Position of p's left child (or null if no child exists). */
  Position<E> left(Position<E> p) throws IllegalArgumentException;
  /** Returns the Position of p's right child (or null if no child exists). */
  Position<E> right(Position<E> p) throws IllegalArgumentException;
  /** Returns the Position of p's sibling (or null if no sibling exists). */
  Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}
```

An AbstractBinaryTree Base Class in Java

```java
/** An abstract base class providing some functionality of the BinaryTree interface.*/
public abstract class AbstractBinaryTree<E> extends AbstractTree<E>
                                             implements BinaryTree<E> {
  /** Returns the Position of p's sibling (or null if no sibling exists). */
  public Position<E> sibling(Position<E> p) {
    Position<E> parent = parent(p);
    if (parent == null) return null;                    // p must be the root
    if (p == left(parent))                              // p is a left child
      return right(parent);                             // (right child might be null)
    else                                                // p is a right child
      return left(parent);                              // (left child might be null)
  }
  /** Returns the number of children of Position p. */
  public int numChildren(Position<E> p) {
    int count=0;
    if (left(p) != null)
      count++;
    if (right(p) != null)
      count++;
    return count;
  }
  /** Returns an iterable collection of the Positions representing p's children. */
  public Iterable<Position<E>> children(Position<E> p) {
    List<Position<E>> snapshot = new ArrayList<>(2);  // max capacity of 2
    if (left(p) != null)
      snapshot.add(left(p));
    if (right(p) != null)
      snapshot.add(right(p));
    return snapshot;
  }
}
```

Properties of Binary Trees

We denote the set of all nodes of a tree T at the same depth d as **level** d of T

level 0: ≤ 1 node;   level 1: ≤ 2 nodes;   ...   level d:

Properties of Binary Trees

We denote the set of all nodes of a tree T at the same depth d as **level** d of T

level 0: ≤ 1 node; level 1: ≤ 2 nodes; ... level d: $\leq 2^d$ nodes

n, the number of nodes in T

n_E, the number of external nodes of T

n_I, the number of internal nodes of T

h, the height of T

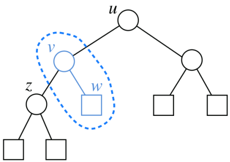
T, a nonempty binary tree	T, a nonempty **proper** binary tree
$h+1 \leq n \leq 2^{h+1}-1$	$2h+1 \leq n \leq 2^{h+1}-1$
$1 \leq n_E \leq 2^h$	$h+1 \leq n_E \leq 2^h$
$h \leq n_I \leq 2^h - 1$	$h \leq n_I \leq 2^h - 1$
$\log(n+1)-1 \leq h \leq n-1$	$\log(n+1)-1 \leq h \leq (n-1)/2$
	$n_E = n_I + 1$

Justification of $n_E = n_I + 1$ for Proper Binary Trees

Separate the nodes from T into an internal-node pile and an external-node pile, until T becomes empty

Case 1: $n = 1$ Place the only node on the external-node pile. Thus, $n_E = n_I + 1$ if $n_E = 1$ and $n_I = 0$.

Case 2: $n > 1$ Place an external node w and its parent v on the external-node pile and the internal-node pile, respectively. If v has a parent u, reconnect u with the former sibling z of w. Repeating this operation, we eventually get Case 1. Thus, the external-node pile has one more node than the internal-node pile.



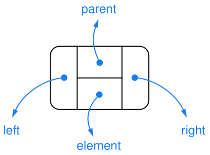
(a) (b) (c)

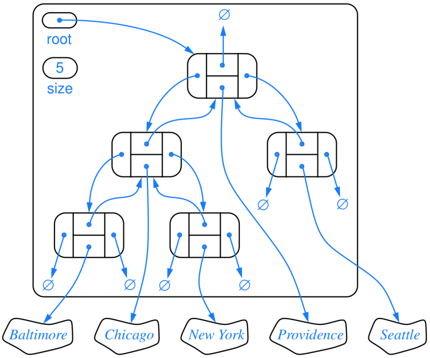
Linked Structure for Binary Trees

A node in a **linked structure** is represented by an object storing

- Element
- Parent node
- Left child nodes
- Right child node

Node objects implement the Position ADT





Updating a Linked Binary Tree

For a linked binary tree, the following **update** methods can be supported (based on *efficiency*):

addRoot(e): Creates a root for an empty tree, storing e as the element, and returns the position of that root; an error occurs if the tree is not empty

addLeft(p, e): Creates a left child of position p, storing element e, and returns the position of the new node; an error occurs if p already has a left child

addRight(p, e): Creates a right child of position p, storing element e, and returns the position of the new node; an error occurs if p already has a right child

set(p, e): Replaces the element stored at position p with element e, and returns he previously stored element

attach(p, T_1, T_2): Attaches the internal structure of trees T_1 and T_2 as the respective left and right subtrees of leaf position p and resets T_1 and T_2 to empty trees; an error condition occurs if p is not a leaf

remove(p): Removes the node at position p, replacing it with its child (if any), and returns the element that had been stored at p; an error occurs if p has two children

```java
/** Concrete implementation of a binary tree using a node-based, linked structure. */
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {

  //---------------- nested Node class ----------------
  protected static class Node<E> implements Position<E> {
    private E element;                 // an element stored at this node
    private Node<E> parent;            // a reference to the parent node (if any)
    private Node<E> left;              // a reference to the left child (if any)
    private Node<E> right;             // a reference to the right child (if any)
    /** Constructs a node with the given element and neighbors. */
    public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {
      element = e;
      parent = above;
      left = leftChild;
      right = rightChild;
    }
    // accessor methods
    public E getElement() { return element; }
    public Node<E> getParent() { return parent; }
    public Node<E> getLeft() { return left; }
    public Node<E> getRight() { return right; }
    // update methods
    public void setElement(E e) { element = e; }
    public void setParent(Node<E> parentNode) { parent = parentNode; }
    public void setLeft(Node<E> leftChild) { left = leftChild; }
    public void setRight(Node<E> rightChild) { right = rightChild; }
  } //----------- end of nested Node class -----------

  /** Factory function to create a new node storing element e. */
  protected Node<E> createNode(E e, Node<E> parent,
                                  Node<E> left, Node<E> right) {
    return new Node<E>(e, parent, left, right);
  }

  // LinkedBinaryTree instance variables
  protected Node<E> root = null;       // root of the tree
  private int size = 0;                // number of nodes in the tree

  // constructor
  public LinkedBinaryTree() { }        // constructs an empty binary tree
```

```
// nonpublic utility
/** Validates the position and returns it as a node. */
protected Node<E> validate(Position<E> p) throws IllegalArgumentException {
  if (!(p instanceof Node))
    throw new IllegalArgumentException("Not valid position type");
  Node<E> node = (Node<E>) p;                    // safe cast
  if (node.getParent( ) == node)                 // our convention for defunct node
    throw new IllegalArgumentException("p is no longer in the tree");
  return node;
}

// accessor methods (not already implemented in AbstractBinaryTree)
/** Returns the number of nodes in the tree. */
public int size( ) {
  return size;
}

/** Returns the root Position of the tree (or null if tree is empty). */
public Position<E> root( ) {
  return root;
}

/** Returns the Position of p's parent (or null if p is root). */
public Position<E> parent(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return node.getParent( );
}

/** Returns the Position of p's left child (or null if no child exists). */
public Position<E> left(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return node.getLeft( );
}

/** Returns the Position of p's right child (or null if no child exists). */
public Position<E> right(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return node.getRight( );
}
```

```
// update methods supported by this class
/** Places element e at the root of an empty tree and returns its new Position. */
public Position<E> addRoot(E e) throws IllegalStateException {
  if (!isEmpty()) throw new IllegalStateException("Tree is not empty");
  root = createNode(e, null, null, null);
  size = 1;
  return root;
}

/** Creates a new left child of Position p storing element e; returns its Position. */
public Position<E> addLeft(Position<E> p, E e)
                        throws IllegalArgumentException {
  Node<E> parent = validate(p);
  if (parent.getLeft( ) != null)
    throw new IllegalArgumentException("p already has a left child");
  Node<E> child = createNode(e, parent, null, null);
  parent.setLeft(child);
  size++;
  return child;
}

/** Creates a new right child of Position p storing element e; returns its Position. */
public Position<E> addRight(Position<E> p, E e)
                        throws IllegalArgumentException {
  Node<E> parent = validate(p);
  if (parent.getRight( ) != null)
    throw new IllegalArgumentException("p already has a right child");
  Node<E> child = createNode(e, parent, null, null);
  parent.setRight(child);
  size++;
  return child;
}

/** Replaces the element at Position p with e and returns the replaced element. */
public E set(Position<E> p, E e) throws IllegalArgumentException {
  Node<E> node = validate(p);
  E temp = node.getElement( );
  node.setElement(e);
  return temp;
}
```

```
  /** Attaches trees t1 and t2 as left and right subtrees of external p. */
  public void attach(Position<E> p, LinkedBinaryTree<E> t1,
                    LinkedBinaryTree<E> t2) throws IllegalArgumentException {
    Node<E> node = validate(p);
    if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");
    size += t1.size() + t2.size();
    if (!t1.isEmpty()) {                    // attach t1 as left subtree of node
      t1.root.setParent(node);
      node.setLeft(t1.root);
      t1.root = null;
      t1.size = 0;
    }
    if (!t2.isEmpty()) {                    // attach t2 as right subtree of node
      t2.root.setParent(node);
      node.setRight(t2.root);
      t2.root = null;
      t2.size = 0;
    }
  }
  /** Removes the node at Position p and replaces it with its child, if any. */
  public E remove(Position<E> p) throws IllegalArgumentException {
    Node<E> node = validate(p);
    if (numChildren(p) == 2)
      throw new IllegalArgumentException("p has two children");
    Node<E> child = (node.getLeft() != null ? node.getLeft() : node.getRight() );
    if (child != null)
      child.setParent(node.getParent());    // child's grandparent becomes its parent
    if (node == root)
      root = child;                         // child becomes root
    else {
      Node<E> parent = node.getParent();
      if (node == parent.getLeft())
        parent.setLeft(child);
      else
        parent.setRight(child);
    }
    size--;
    E temp = node.getElement();
    node.setElement(null);                  // help garbage collection
    node.setLeft(null);
    node.setRight(null);
    node.setParent(node);                   // our convention for defunct node
    return temp;
  }
} //----------- end of LinkedBinaryTree class -----------
```

Linked Binary Tree: Analysis

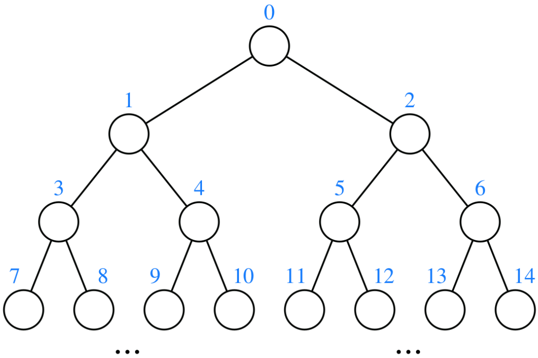
Method	Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

Space usage: $O(n)$, where n is the number of nodes in the tree

Numbering the Positions of a Binary Tree

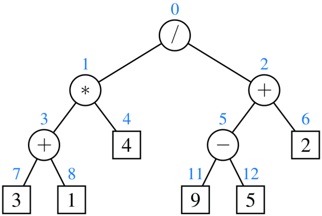
The function f is called **level numbering** of positions in T

- If p is the root, then $f(p) = 0$
- If p is the left child of q, then $f(p) = 2f(q) + 1$
- If p is the right child of q, then $f(p) = 2f(q) + 2$



Array-Based Representation of a Binary Tree

When representing a binary tree T by means of an array-based structure A, we store the element at position p of T at index $f(p)$, i.e. element of position p is stored at $A[f(p)]$



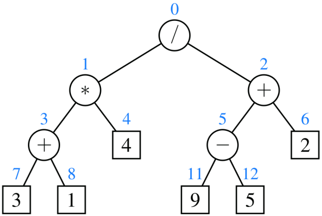
/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Advantage: a position p can be represented by the one integer $f(p)$

The left child of p has index

Array-Based Representation of a Binary Tree

When representing a binary tree T by means of an array-based structure A, we store the element at position p of T at index $f(p)$, i.e. element of position p is stored at $A[f(p)]$



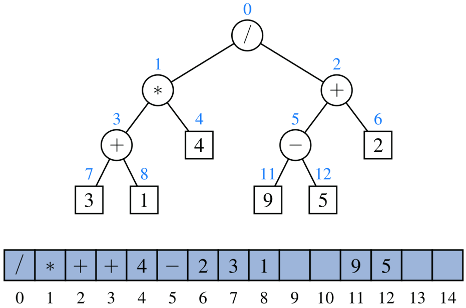
/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Advantage: a position p can be represented by the one integer $f(p)$

The left child of p has index $2f(p) + 1$

Array-Based Representation of a Binary Tree

When representing a binary tree T by means of an array-based structure A, we store the element at position p of T at index $f(p)$, i.e. element of position p is stored at $A[f(p)]$



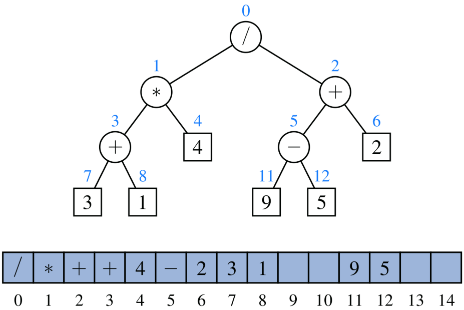
Advantage: a position p can be represented by the one integer $f(p)$

The left child of p has index $2f(p) + 1$

The right child of p has index

Array-Based Representation of a Binary Tree

When representing a binary tree T by means of an array-based structure A, we store the element at position p of T at index $f(p)$, i.e. element of position p is stored at $A[f(p)]$



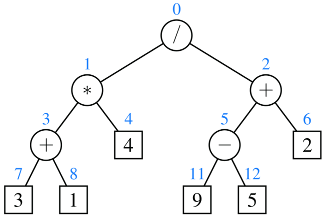
Advantage: a position p can be represented by the one integer $f(p)$

The left child of p has index $2f(p) + 1$

The right child of p has index $2f(p) + 2$

Array-Based Representation of a Binary Tree

When representing a binary tree T by means of an array-based structure A, we store the element at position p of T at index $f(p)$, i.e. element of position p is stored at $A[f(p)]$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
/	*	+	+	4	−	2	3	1			9	5		

Advantage: a position p can be represented by the one integer $f(p)$

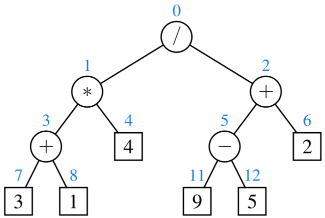
The left child of p has index $2f(p) + 1$

The right child of p has index $2f(p) + 2$

The parent of p has index

Array-Based Representation of a Binary Tree

When representing a binary tree T by means of an array-based structure A, we store the element at position p of T at index $f(p)$, i.e. element of position p is stored at $A[f(p)]$



/	*	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Advantage: a position p can be represented by the one integer $f(p)$

The left child of p has index $2f(p) + 1$

The right child of p has index $2f(p) + 2$

The parent of p has index $\lfloor (f(p) - 1)/2 \rfloor$

Properties of Array-Based Binary Tree

Let n be the number of nodes of T, and let f_M be the maximum value of $f(p)$ over all the nodes of T

The array A requires length $N = 1 + f_M$, since elements range from $A[0]$ to $A[f_M]$

Note that A may have a number of empty cells that do not refer to existing positions of T

In the worst case, $N = 2^n - 1$

Why? Can you construct such a binary tree?

Later we will see applications for which the array representation of a binary tree is space efficient

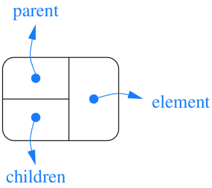
Another drawback: many update operations like removing a node and promoting its child takes $O(n)$ time since all descendants of that child move locations.

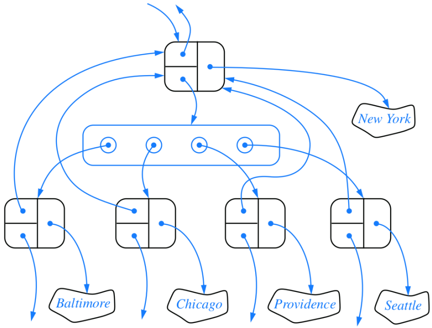
Linked Structure for Trees

A node in a **linked structure** is represented by an object storing

- Element
- Parent node
- Sequence of children nodes

Node objects implement the Position ADT





Summary

Reading

Section 8.2 Binary Trees

Section 8.3 Implementing Trees

Questions?