CS121 Data Structures A, C
Trees

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Trees

All the data structures considered in the first half of the course were **linear** (simple ‘before’ and ‘after’ relationships between objects in sequences)

Now, we turn to **nonlinear** data structures

Trees are one of the most important nonlinear data structures in computing

The relationships in a tree are **hierarchical**, with some objects being ‘above’ and some ‘below’ others

Similar to family trees, the terms are ‘parent’, ‘child’, ‘ancestor’, ‘descendant’, etc.

Tree Definition

A **tree** is an abstract data type that stores elements hierarchically

Each element in a tree has a **parent** element (except the top element called the **root**) and zero or more **children** elements

Formally, we define a **tree** T as a set of **nodes** storing elements such that the nodes have a **parent–child** relationship that satisfies:

- If T is nonempty, it has a special node, called the **root** of T, that has no parent
- Each node v of T different from the root has a unique **parent** node w; every node with parent w is a **child** of w

A tree can be empty, meaning that it does not have any nodes

A tree T is either empty or consists of a root node r and a (possibly empty) set of subtrees whose roots are the children of r

Node Relationships

Two nodes that are children of the same parent are **siblings**

A node v is **external** (also known as a **leaf**) if v has no children

A node v is **internal** if it has one or more children

A node u is an **ancestor** of a node v if $u = v$ or u is an ancestor of the parent of v

Conversely, a node v is a **descendant** of a node u if u is an ancestor of v

The **subtree** of T **rooted** at a node v is the tree consisting of all the descendants of v in T (including v itself)

Edges and Paths

An **edge** of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa

A **path** of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge

Ordered Trees

A tree is **ordered** if there is a meaningful linear order among the children of each node

That is, we purposefully identify the children of a node as being the first, second, third, and so on

Usually visualized by arranging siblings left to right, according to their order

The Position Abstract Data Type

We rely on the concept of a **position** as an abstraction for a tree node storing an element and satisfying parent–child relationships

A position supports the following single method:

getElement(): Returns (a reference to) the element stored at this position

The Tree Abstract Data Type

The tree ADT supports **accessor** methods:

root():	Returns the position of the root of the tree (or `null` if empty)
parent(p):	Returns the position of the parent of position p (or `null` if p is the root)
children(p):	Returns an iterable collection containing the children of position p (if any)
numChildren(p):	Returns the number of children of position p

If a tree T is ordered, then `children(`p`)` reports the children of p in order

The tree ADT supports **query** methods:

isInternal(p):	Returns `true` if position p has at least one child
isExternal(p):	Returns `true` if position p does not have any children
isRoot(p):	Returns `true` if position p is the root of the tree

The Tree Abstract Data Type (cont'd)

The tree ADT supports **general** methods:

size():	Returns the number of positions (and hence elements) that are contained in the tree
isEmpty():	Returns `true` if the tree does not contain any positions (and thus no elements)
iterator():	Returns an iterator for all elements in the tree (so that the tree itself is `Iterable`)
positions():	Returns an iterable collection of all positions of the tree

If an invalid position is sent as a parameter to any method of a tree, then an error is generated

A Tree Interface in Java

We rely upon the same definition of the `Position` interface as introduced for positional lists

```
/** An interface for a tree where nodes can have an arbitrary number of children. */
public interface Tree<E> extends Iterable<E> {
  Position<E> root( );
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isExternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size( );
  boolean isEmpty( );
  Iterator<E> iterator( );
  Iterable<Position<E>> positions( );
}
```

An AbstractTree Base Class in Java

We provide concrete implementations for some of the methods in an abstract class named `AbstractTree`

```java
/** An abstract base class providing some functionality of the Tree interface. */
public abstract class AbstractTree<E> implements Tree<E> {
  public boolean isInternal(Position<E> p) { return numChildren(p) > 0; }
  public boolean isExternal(Position<E> p) { return numChildren(p) == 0; }
  public boolean isRoot(Position<E> p) { return p == root( ); }
  public boolean isEmpty( ) { return size( ) == 0; }
}
```

Depth and Height

The **depth** of a position p within tree T is the number of ancestors of p, other than p itself

This implies that the depth of the root of T is 0

A recrusive definition of depth:

- If p is the root, then the depth of p is 0
- Otherwise, the depth of p is one plus the depth of the parent of p

The **height** of a tree equals to the maximum of the depths of its positions (or zero, if the tree is empty)

It is easy to see that the position with maximum depth must be a leaf

Computing Depth

```
/** Returns the number of levels separating Position p from the root. */
public int depth(Position<E> p) {
  if (isRoot(p))
    return 0;
  else
    return 1 + depth(parent(p));
}
```

The running time of `depth` for position p is

Computing Depth

```
/** Returns the number of levels separating Position p from the root. */
public int depth(Position<E> p) {
  if (isRoot(p))
    return 0;
  else
    return 1 + depth(parent(p));
}
```

The running time of `depth` for position p is $O(d_p + 1)$, where d_p denotes the depth of p in the tree. Thus, it runs in $O(n)$ worst-case time, where n is the total number of positions of T.

Computing Height, First Approach

```
/** Returns the height of the tree. */
private int heightBad( ) {                         // works, but quadratic worst-case time
  int h = 0;
  for (Position<E> p : positions( ))
    if (isExternal(p))                             // only consider leaf positions
      h = Math.max(h, depth(p));
  return h;
}
```

If `positions()` runs is $O(n)$, then `heightBad` runs in

Computing Height, First Approach

```
/** Returns the height of the tree. */
private int heightBad( ) {                    // works, but quadratic worst-case time
  int h = 0;
  for (Position<E> p : positions( ))
    if (isExternal(p))                        // only consider leaf positions
      h = Math.max(h, depth(p));
  return h;
}
```

If `positions()` runs is $O(n)$, then `heightBad` runs in $O(n + \sum_{p \in L}(d_p + 1))$, where L is the set of leaf positions of T. In the worst case, $\sum_{p \in L}(d_p + 1)$ is proportional to n^2, thus `heightBad` runs in $O(n^2)$ worst-case time

Computing Height, Better Approach

We define the **height** of a position p in a tree T as follows:

- If p is a leaf, then the height of p is 0
- Otherwise, the height of p is one more than the maximum of the heights of p's children

The height of the root of a nonempty tree T, according to this definition, equals the maximum depth among all leaves of tree T

```
/** Returns the height of the subtree rooted at Position p. */
public int height(Position<E> p) {
  int h = 0;                              // base case if p is external
  for (Position<E> c : children(p))
    h = Math.max(h, 1 + height(c));
  return h;
}
```

Computing Height, Better Approach (cont'd)

We assume that an iteration on `children(p)` is executed in $O(c_p + 1)$ time, where c_p denotes the number of children of p

`height(p)` is called once recursively for all positions of T

The overall running time is $O(\sum_p (c_p + 1)) = O(n + \sum_p c_p)$

Let T be a tree with n positions, and let c_p denote the number of children of a position p of T. Then, summing over the positions of T, $\sum_p c_p = n - 1$

This is true since each position of T, except the root, is a child of another position, and thus contributes one unit to the above sum

Thus, the running time of *height*, when called on the root of T, is $O(n)$, where n is the number of positions of T

Summary

Reading

Section 8.1 General Trees

Questions?