CSE221 Data Structures
Lecture 13: Trees

Antoine Vigneron
antoine@unist.ac.kr

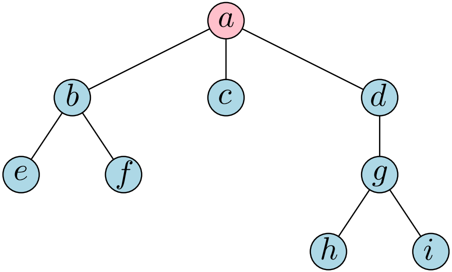
Ulsan National Institute of Science and Technology

November 1, 2021

Introduction

- I updated attendance records. They can be found in the portal under E-attendance.
- I will grade the midterm this week.
- Reference for this lecture: Textbook Section 7.1-7.2

Trees

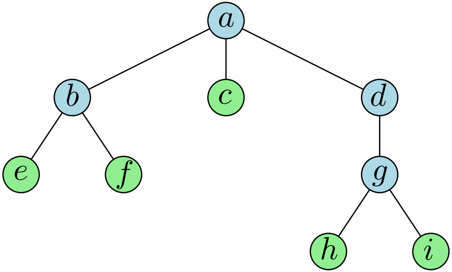


- A *tree* T with 9 *nodes*: a, b, c, d, e, f, g, h, i.
- Its *root* is a.
- d is the *parent* of g.
- h and i are the *children* of g.

Trees

- A tree is an abstract data type that stores elements hierarchically.
- Elements are stored at nodes.
- Each node has exactly one parent, except for one node called the root.
- If u is the parent of v, then v is a child of u.
- Trees are usually drawn from top to bottom, the root being on top.

Trees

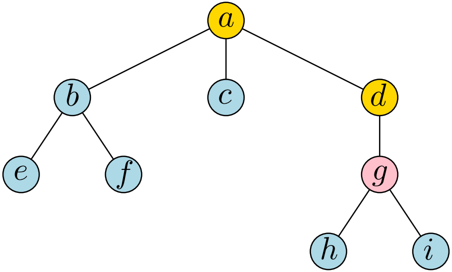


- The nodes that have children are called *internal nodes* (blue).
- The nodes that have no children are called *leaves*, or *external nodes* (green).
- Two nodes that have the same parent are called *siblings*.

Example

b and c are siblings.

Trees



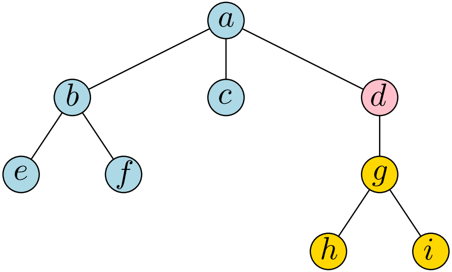
- The *ancestors* of a node are its parent, and the ancestors of its parent.

Example

The ancestors of g are a and d.

- Remark: In the textbook, a node is also considered to be an ancestor of itself. We will rather use the standard convention that it is not.

Trees

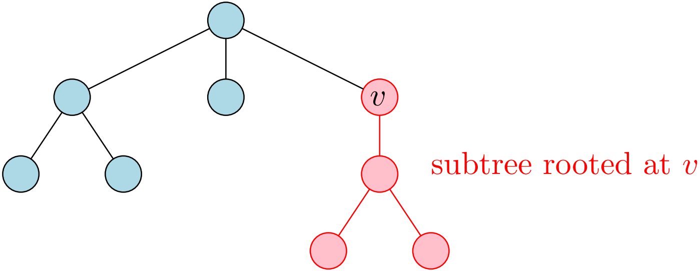


- The *descendants* of a node are its children, or the descendants of its children.
- In other words, v is a descendant of u iff u is an ancestor of v.

Example

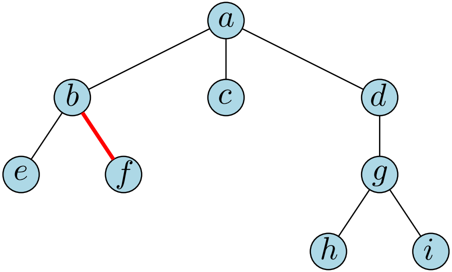
The descendants of d are g, h and i.

Trees



- The *subtree* of T rooted at a node v is the tree consisting of v and all the descendants of v in T.

Trees

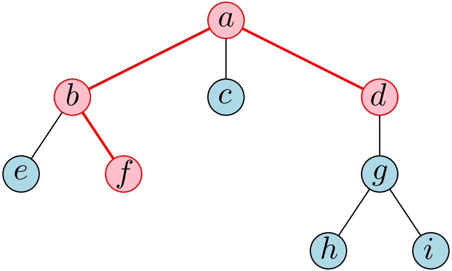


- An *edge* is a pair of nodes (u, v) such that u is the parent of v, or v is the parent of u.

Example

The edge (b, f) is shown in red.

Trees

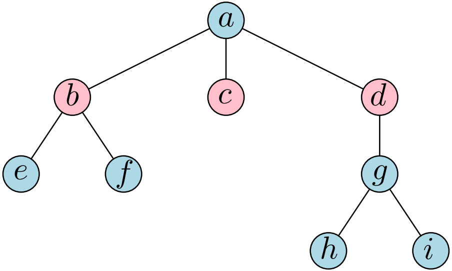


- A *path* from a node s to a node t is a sequence of nodes such that any two consecutive nodes form an edge, the first node is s and the last node is t.

Example

The path (f, b, a, d) is a path from f to d.

Ordered Trees

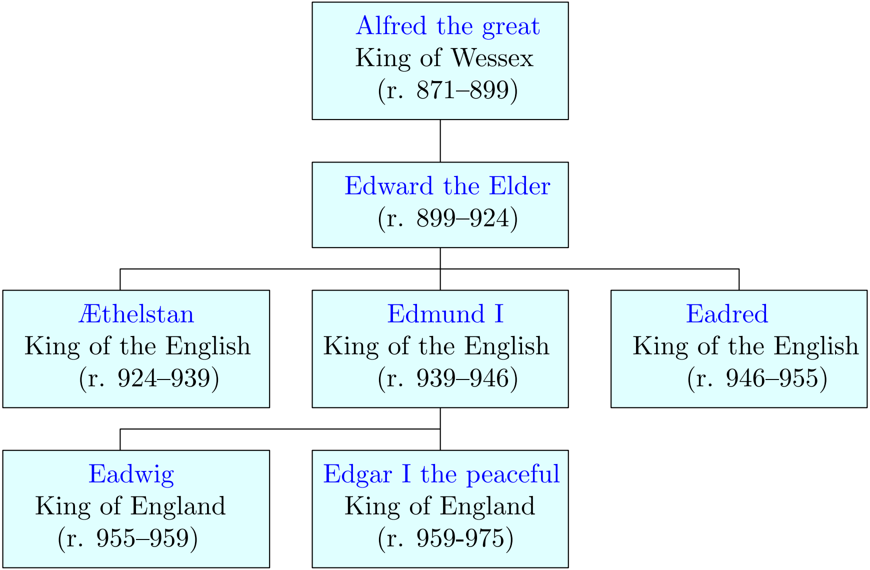


- In an *ordered tree*, the children of each node are given in a specific order. They are usually drawn from left to right.

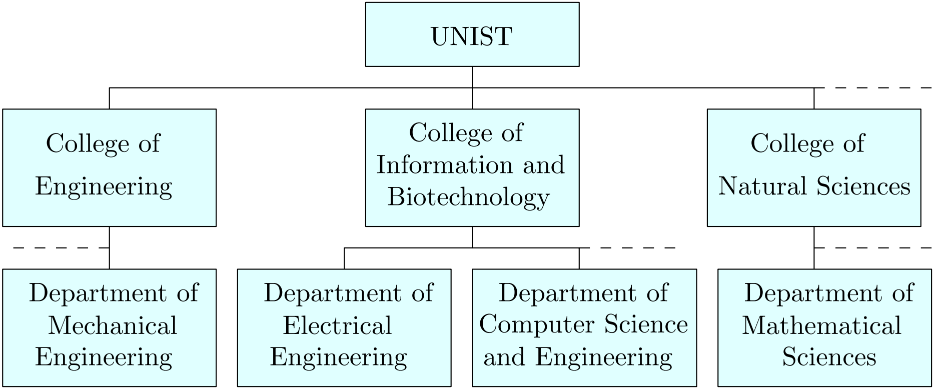
Example

In the tree above, we use alphabetical order. The three children of a are given in the order b, c, d.

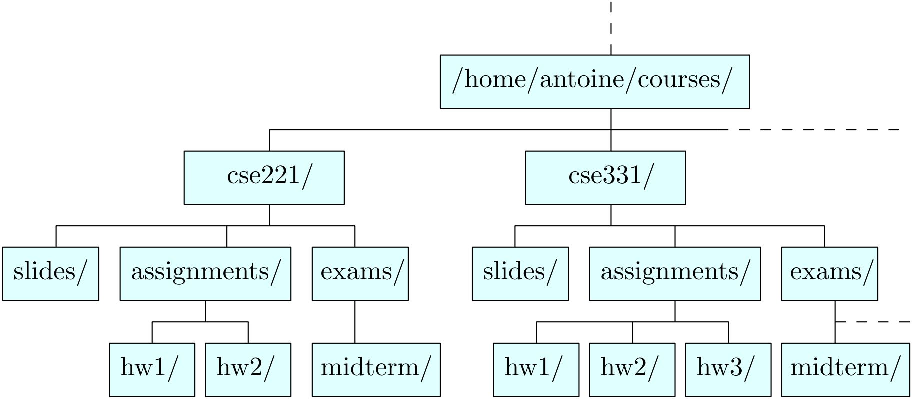
Example: A Family Tree



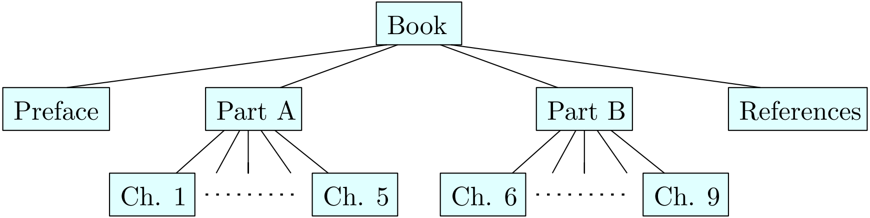
Example: A University



Example: A File System



Example



Tree Functions

- Each node of the tree T is associated with a *position* object p.
- The deferencing operator $*$ is overloaded, so the element stored at this node is given by $*p$.
- A *position list* is a list whose elements are tree positions.

- p.**parent**(): Return the parent of p; an error occurs if p is the root.
- p.**children**(): Return a position list containing the children of node p.
- p.**isRoot**(): Return true if p is the root and false otherwise.
- p.**isExternal**(): Return true if p is external and false otherwise.

- 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

```
template <typename E>                          // base element type
class Position<E> {                            // a node position
public:
  E& operator*();                              // get element
  Position parent() const;                     // get parent
  PositionList children() const;   // get node's children
  bool isRoot() const;                         // root node?
  bool isExternal() const;                     // external node?
};
```

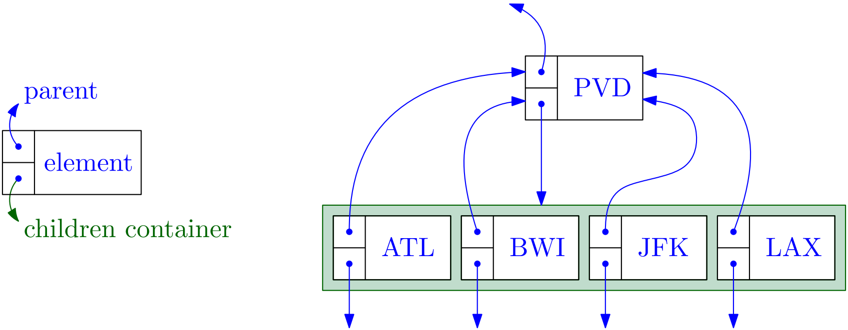
Class representing a position in the tree

C++ Interface

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

- `PositionList` is a list of position, for instance it could be `std::list<Position>`.

Linked Structure

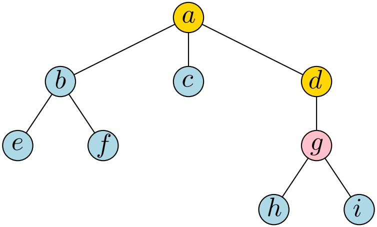


Linked Structure

Operation	Time
`isRoot()`, `isExternal()`	$O(1)$
`parent()`	$O(1)$
`children(p)`	$O(1 + c_p)$
`size()`, `empty()`	$O(1)$
`root()`	$O(1)$
`positions()`	$O(n)$

- c_p is the number of children of the node at position p.
- n is the number of nodes in the tree.

Depth of a node



- The number of ancestors of a node p is its *depth*.

Examples

- $\text{depth}(g) = 2$
- The depth of the root is $\text{depth}(a) = 0$.

Depth of a Node

- The depth of a node can also be defined recursively:

$$\text{depth}(p) = \begin{cases} 0 & \text{if } p \text{ is the root, and} \\ 1+\text{depth}(\text{parent}(p)) & \text{otherwise.} \end{cases}$$

- It suggests the following algorithm for computing it:

Pseudocode

```
procedure DEPTH(T, p)
    if p.isRoot() then
        return 0
    return 1+depth(T,parent(p))
```

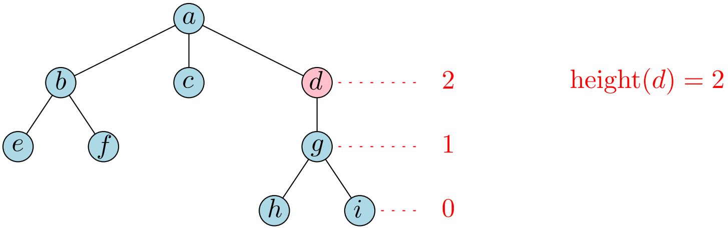
Depth of a Node

C++ Code

```
int depth(const Tree& T, const Position& p) {
  if (p.isRoot())
    return 0;                          // root has depth 0
  return 1 + depth(T, p.parent());  // 1+(depth of parent)
}
```

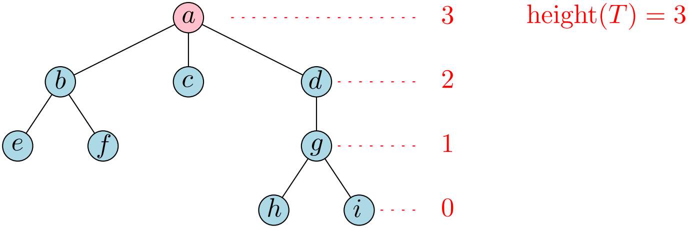
- Remark: This is a recursive algorithm. It will often be the case with tree algorithms: The simplest implementation is recursive.
- Running time: $O(1 + d_p)$ where d_p is the depth of p.
- So in the worst case, it could be $\Theta(n)$.

Height of a Tree



- If a node p is a leaf, then its *height* is 0.
- Otherwise, it is one plus the maximum height of a child of p.

Height of a Tree



- The height of a tree is the height of its root. Alternatively:

Proposition

The height of a tree is equal to the maximum depth of its leaves.

Pseudocode

procedure HEIGHT1(T)
  $h \leftarrow 0$
  for each $p \in T$.positions() **do**
    if p.isExternal() **then**
      $h \leftarrow \max(h, \text{depth}(T, p))$
  return h

C++ Code

```
int height1(const Tree& T) {
  int h = 0;
  PositionList nodes = T.positions(); // list of all nodes
  for (Iterator q = nodes.begin(); q != nodes.end(); ++q)
    if (q->isExternal())                   // get max depth
      h = max(h, depth(T, *q));            // among leaves
  return h;
}
```

Height of a Tree

- What is the running time of `height1()`?
- Answer:

$$T(n) = O\left(n + \sum_{p \in L}(1 + d_p)\right)$$

  where L is the set of leaves (external nodes) of the tree.
- In the worst case, this is $\Theta(n^2)$.

(Exercise C-7.8, done in class.)

Height of a Tree

- We now present a better algorithm.

Pseudocode

```
procedure HEIGHT2(T)
    if p.isExternal() then
        return 0
    h ← 0
    for each q ∈ p.children() do
        h ← max(h, height2(T, q))
    return 1+h
```

C++ Code

```cpp
int height2(const Tree& T, const Position& p) {
  if (p.isExternal()) return 0;       // leaf has height 0
  int h = 0;
  PositionList ch = p.children();      // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q)
    h = max(h, height2(T, *q));
  return 1 + h;                  // 1 + max height of children
}
```

Analysis

- `height2` is a recursive algorithm, that is called once at each node of the tree.
- At each node p, the **for** loop is iterated c_p times.
- The rest of the code takes $O(1)$ time.
- So the running time is

$$T(n) = O\left(\sum_{p \in V} 1 + c_p\right) = O\left(n + \sum_{p \in V} c_p\right)$$

where V is the set of nodes of T.

Analysis

Proposition

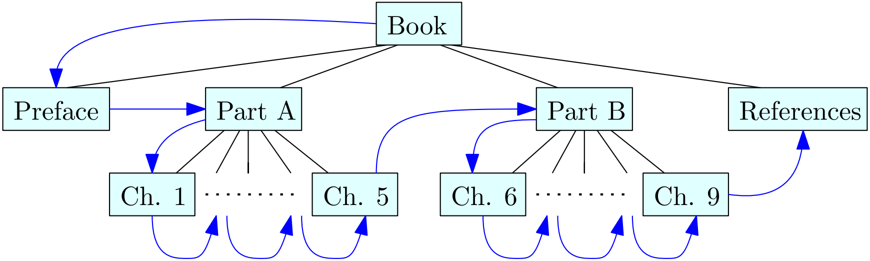
$$\sum_{p \in V} c_p = n - 1$$

Proof.

Each node, except for the root, is the child of exactly one node. $\square$

- It follows that `height2` runs in $O(n)$ time.
- This is much better than `height1` which runs in $O(n^2)$ time.
- Notice that `height2` is *recursive* while `height1` is *iterative*.
- Again, with tree algorithms, it is often better to use recursion.

Preorder Traversal



- The nodes are visited in this order:

  Book, Preface, Part A, Ch. 1, . . . , Ch. 5,
  Part B, Ch. 6, . . . , Ch.9, References.

- This is the table of contents of the book.

Preorder Traversal

- A *traversal* of a tree is a way of visiting all of its nodes.
- In a preorder traversal of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively.
- If the tree is ordered, then the subtrees are traversed according to the order of the children.

Pseudocode

procedure PREORDER(T, p)
    Perform the "visit" action for node p.
    for each child q of p **do**
        `preorder`(T, q)

- The "visit" action is the action that you want to perform on each node of T. For instance, in previous slide, we could print the content of each node, which would print the table of contents of the book.

Analysis

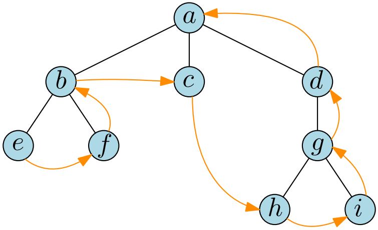
- Suppose that "visit" takes constant time per node.
- `Preorder` is called recursively on each node exactly once.
- Each such call takes $O(1 + c_p)$ time.
- `preorder` takes $O(n)$ time by the same analysis as we did for `height2`.

Preorder Traversal

C++ Code

```
void preorderPrint(const Tree& T, const Position& p) {
  cout << *p;                                 // print element
  PositionList ch = p.children();          // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    cout << " ";
    preorderPrint(T, *q);
  }
}
```

Postorder Traversal



- The nodes are visited in this order:

$$e, f, b, c, h, i, g, d, a$$

Postorder Traversal

Pseudocode

```
procedure PREORDER(T, p)
    for each child q of p do
        preorder(T, q)
    Perform the "visit" action for node p.
```

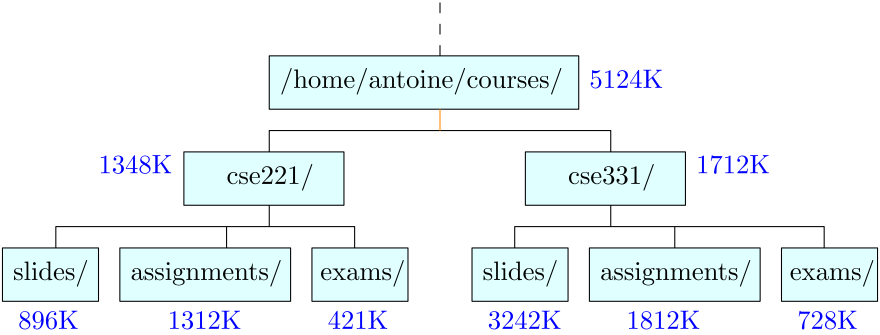
- So we first traverse the subtrees recursively, before we visit the root.
- It also runs in $O(n)$ time, if each visit action takes $O(1)$ time.

Postorder Traversal

C++ Code

```
void postorderPrint(const Tree& T, const Position& p) {
  PositionList ch = p.children();     // list of children
  for (Iterator q = ch.begin(); q != ch.end(); ++q) {
    postorderPrint(T, *q);
    cout << " ";
  }
  cout << *p;                                 // print element
}
```

Postorder Traversal



- Suppose that you want to compute the size of a directory.
- More precisely, this size is the size of the files stored in this directory, plus the size of all its subdirectories.
- We can do it by postorder traversal (see next slide).

Postorder Traversal

C++ Code

```
int diskSpace(const Tree& T, const Position& p) {
  int s = size(p);                    // start with size of p
  if (!p.isExternal()) {                     // if p is internal
    PositionList ch = p.children();// list of p's children
    for (Iterator q = ch.begin(); q != ch.end(); ++q)
      s += diskSpace(T, *q);  // sum the space of subtrees
      cout << name(p) << ": " << s << endl;
                                            // print summary
    }
  return s;
}
```