MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

23 Trees

Department of Computing and Software

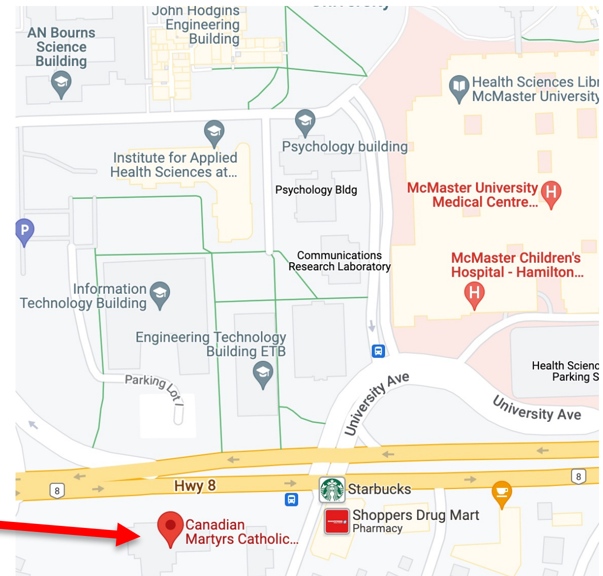
Instructor:

Omid Isfahanialamdari

March 16, 2022

Admin

- Mid-Term 2:
  - Wednesday 23 March 2022
  - Duration: **1 hour**
  - Location: **MCMST CDN_MARTYRS**
    - Seems to be here, I am not sure



  - Covers: Topics from “Doubly Linked Lists” until the lecture of Wednesday 16 March 2022 (inclusive)

Iterating through a Container (Review)

- Let **C** be a **container** and **p** be an **iterator** for **C**

  for (p = C.begin(); p != C.end(); ++p)

  loop_body

- Example: (with an STL vector)
  - Notice how for loop is implemented
  - Notice how we access the element of iterator

```
#include <iostream>
#include <vector>
using namespace std;

int  main(){
    vector<int> v1;
    for (int i = 1; i <= 5; i++)
        v1.push_back(i);

    typedef vector<int>::iterator Iterator;
    int sum = 0;
    for (Iterator p = v1.begin(); p != v1.end(); ++p){
        cout << "Iterator is on data " << *p << endl;
        sum += *p;
    }
    cout << "The sum is :" << sum << endl;

}
```

Output:
Iterator is on data 1
Iterator is on data 2
Iterator is on data 3
Iterator is on data 4
Iterator is on data 5
The sum is :15

Implementing Iterators (remained from pre. lecture)

- Array-based
  - **Array A of the n elements**
  - **index i** that keeps track of the cursor
  - **begin()** = 0
  - **end()** = n (index following the last element)
- Linked list-based
  - For example: A doubly linked-list **L** storing the elements, with sentinels for header and trailer
  - **pointer to node** containing the current element
  - **begin()** = front node (immediately after the header)
  - **end()** = trailer node (immediately after last node)

Implementation of Iterator for (Node) List

- This is an implementation of the iterator for our NodeList (aka DLL)

```
class Iterator {                          // an iterator for the list
public:
  Elem& operator*();                      // reference to the element
  bool operator==(const Iterator& p) const; // compare positions
  bool operator!=(const Iterator& p) const;
  Iterator& operator++();                 // move to next position
  Iterator& operator--();                 // move to previous position
  friend class NodeList;                  // give NodeList access
private:
  Node* v;                                // pointer to the node
  Iterator(Node* u);                      // create from node
};
```

```
struct Node {
  Elem elem;
  Node* prev;
  Node* next;
};
```

```
NodeList::Iterator::Iterator(Node* u)     // constructor from Node*
  { v = u; }

Elem& NodeList::Iterator::operator*()     // reference to the element
  { return v->elem; }
                                          // compare positions
bool NodeList::Iterator::operator==(const Iterator& p) const
  { return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const
  { return v != p.v; }
                                          // move to next position
NodeList::Iterator& NodeList::Iterator::operator++()
  { v = v->next; return *this; }
                                          // move to previous position
NodeList::Iterator& NodeList::Iterator::operator--()
  { v = v->prev; return *this; }
```

(Node) List ADT (duplicate slide)

- This implementation is basically a Doubly Linked-List
- **n**:
  - The number of data elements
- Iterator is an implementation of the Position ADT
  - We use iterator instead of a pointer to a specific node in this implementation

```
typedef int Elem;                                   // list base element type
class NodeList {                                    // node-based list
private:
  // insert Node declaration here...
public:
  // insert Iterator declaration here...
public:
  NodeList();                                       // default constructor
  int size() const;                                 // list size
  bool empty() const;                               // is the list empty?
  Iterator begin() const;                           // beginning position
  Iterator end() const;                             // (just beyond) last position
  void insertFront(const Elem& e);                  // insert at front
  void insertBack(const Elem& e);                   // insert at rear
  void insert(const Iterator& p, const Elem& e); // insert e before p
  void eraseFront();                                // remove first
  void eraseBack();                                 // remove last
  void erase(const Iterator& p);                    // remove p
  // housekeeping functions omitted...
private:                                            // data members
  int     n;                                        // number of items
  Node*   header;                                   // head-of-list sentinel
  Node*   trailer;                                  // tail-of-list sentinel
};
```

STL Iterators in C++

- Each STL container type C supports iterators:
  - **C::iterator** – read/write iterator type
  - **C::const_iterator** – read-only iterator type
  - **C.begin(), C.end()** – return start/end iterators
- This iterator-based operators and methods:
  - ***p:** access current element
  - **++p, --p:** advance to next/previous element
  - **C.assign(p, q):** replace C with contents referenced by the iterator range [p, q) (from p up to, but not including, q)
  - **insert(p, e):** insert e prior to position p
  - **erase(p):** remove element at position p
  - **erase(p, q):** remove elements in the iterator range [p, q)

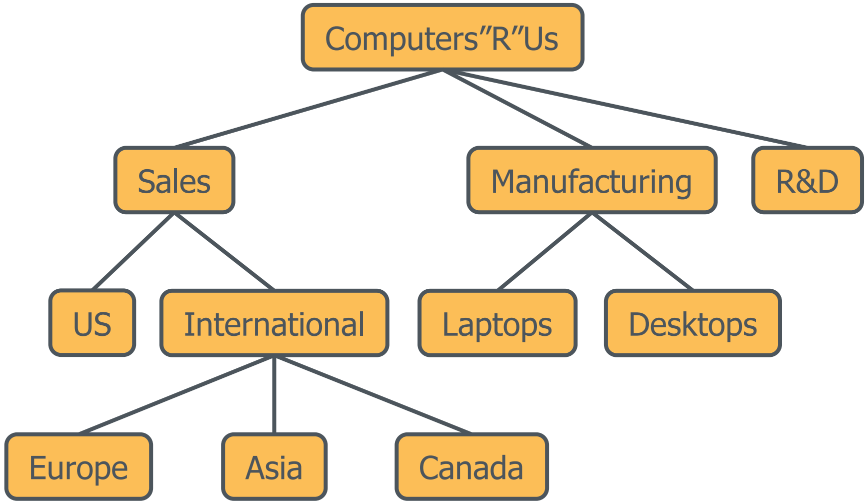
STL List in C++

```
#include <list>
using std::list;                    // make list accessible
list<float> myList;                 // an empty list of floats
```

- list(n): Construct a list with n elements; if no argument list is given, an empty list is created.
- size(): Return the number of elements in L.
- empty(): Return true if L is empty and false otherwise.
- front(): Return a reference to the first element of L.
- back(): Return a reference to the last element of L.
- push_front(e): Insert a copy of e at the beginning of L.
- push_back(e): Insert a copy of e at the end of L.
- pop_front(): Remove the fist element of L.
- pop_back(): Remove the last element of L.

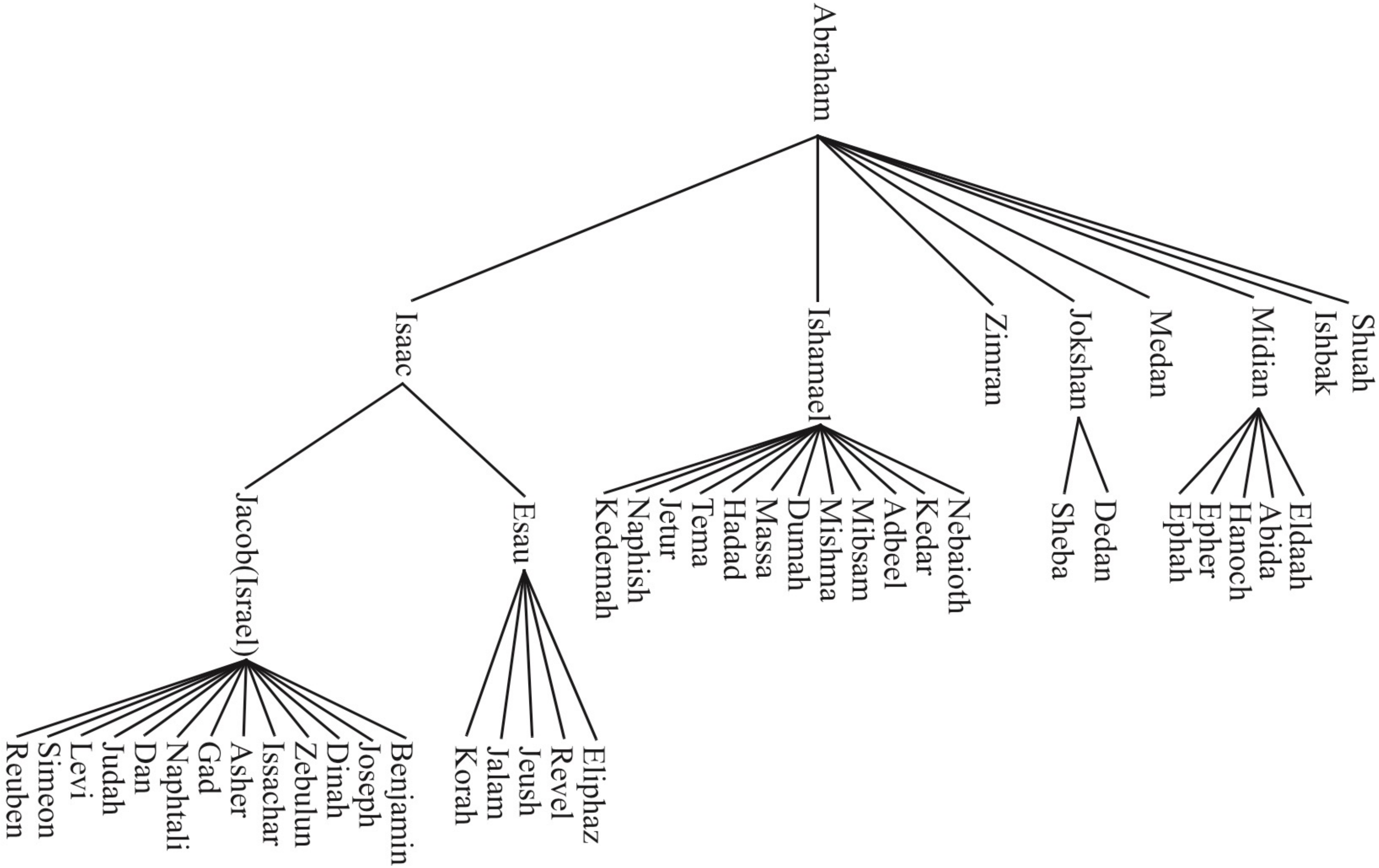
Trees

- So far you are familiar with:
  - Storing elements in a linear fashion
  - Containers and iterators
- In computer science, a tree is an abstract model of a hierarchical structure
- The relation between elements is non-linear
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



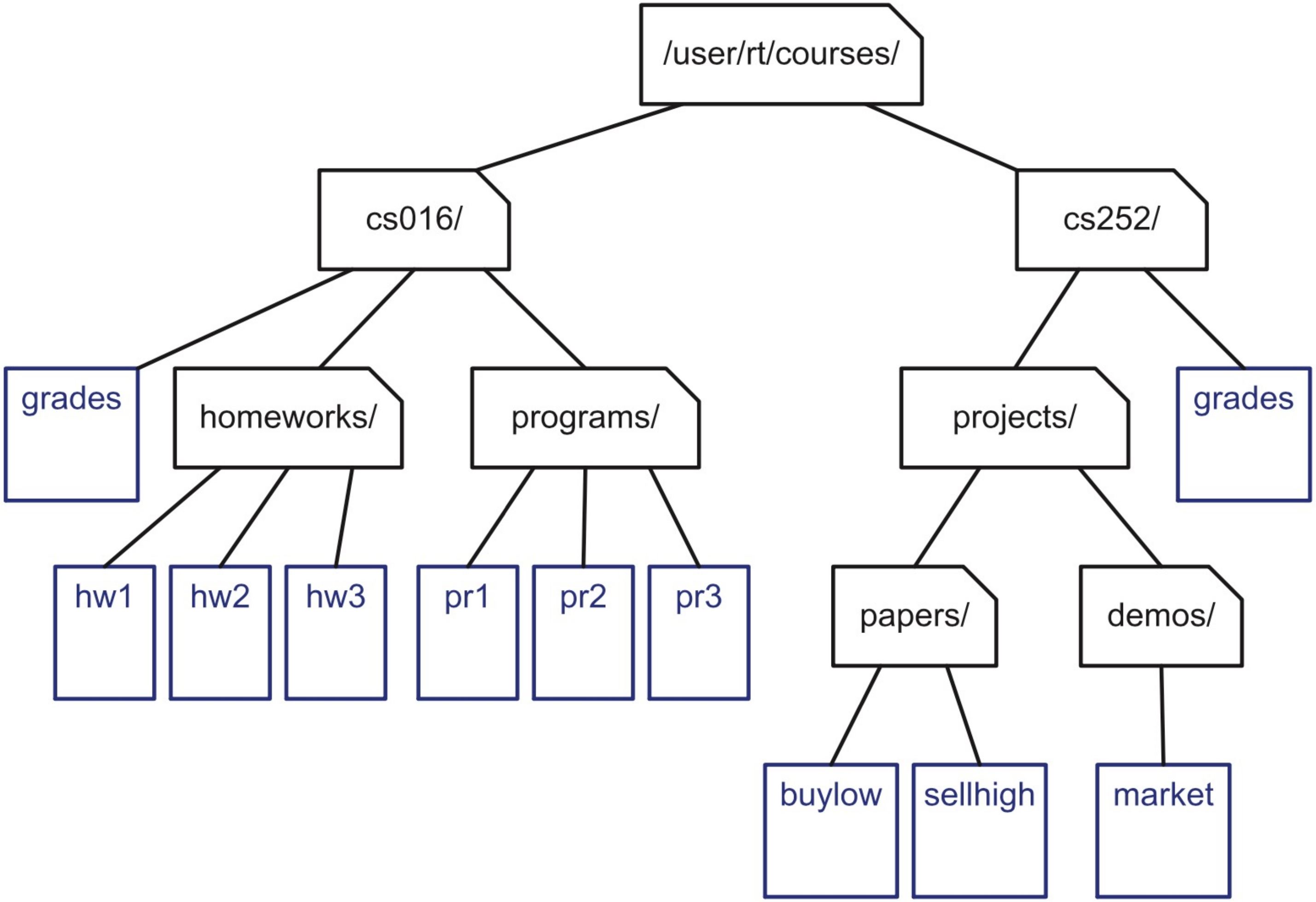
Tree Examples

- Family Tree



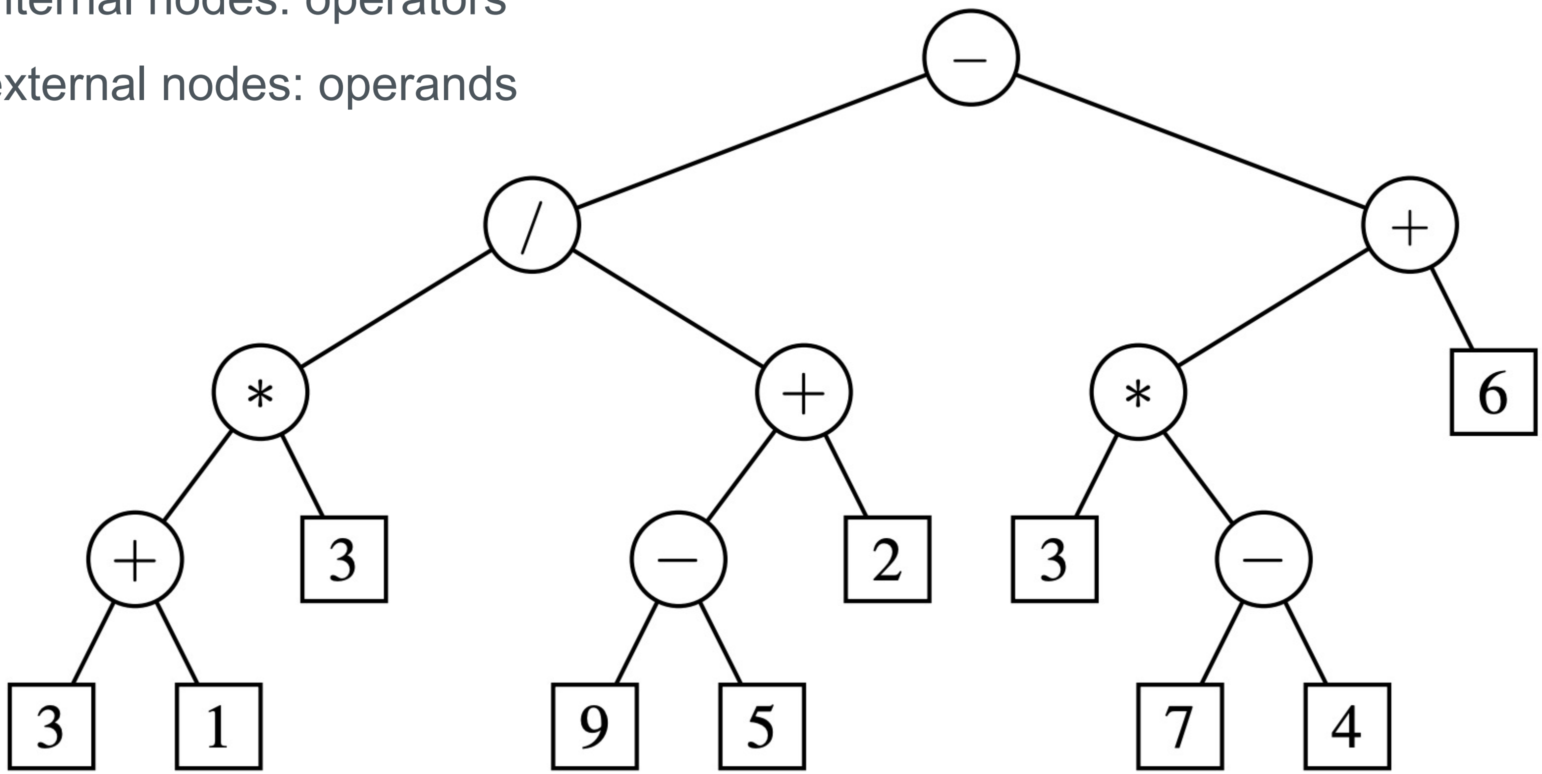
Tree Examples

- File System Tree



Tree Examples

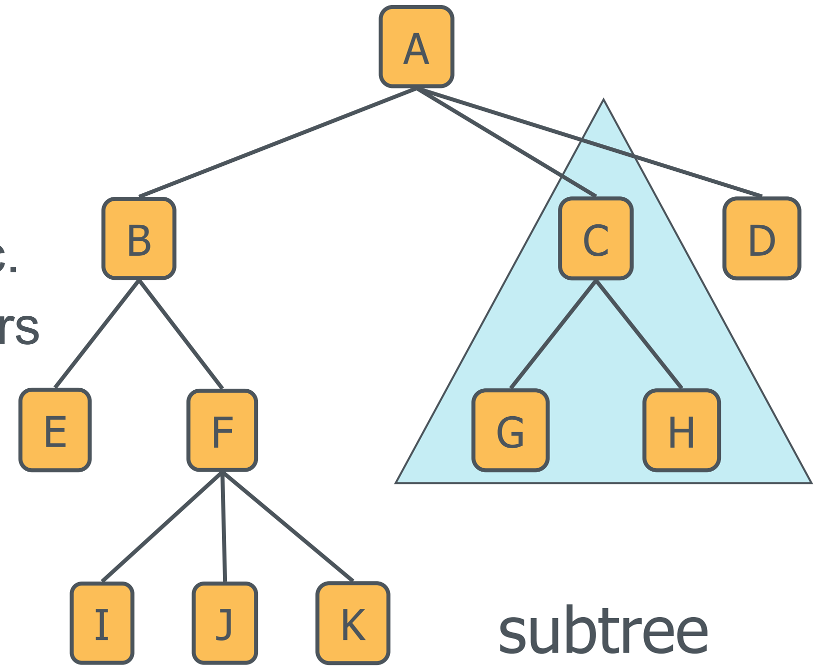
- Expression Tree
  - Binary tree associated with an arithmetic expression
    - internal nodes: operators
    - external nodes: operands



Tree represents: $((((3 + 1) * 3)/((9 - 5) + 2)) - ((3 * (7 - 4)) + 6))$

Trees

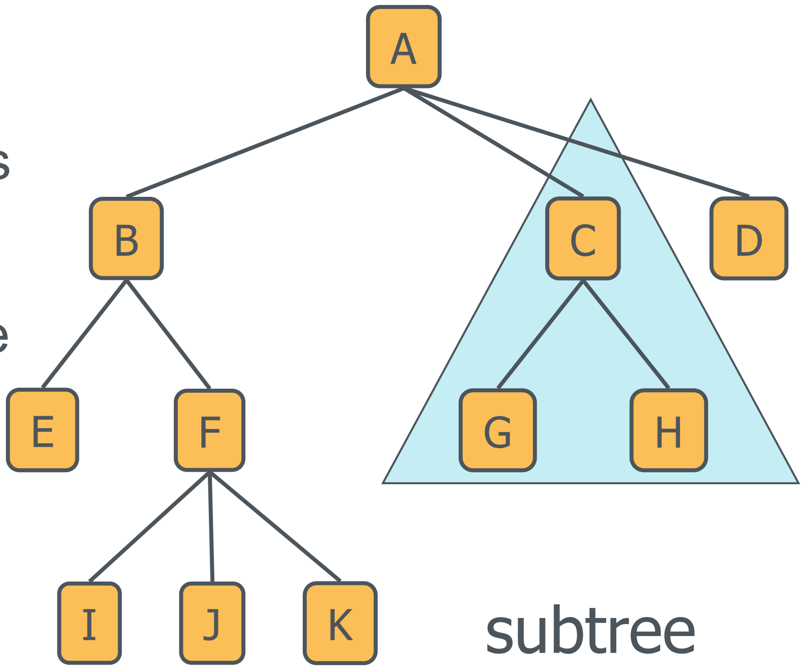
- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.



- Subtree: tree consisting of a node and its descendants

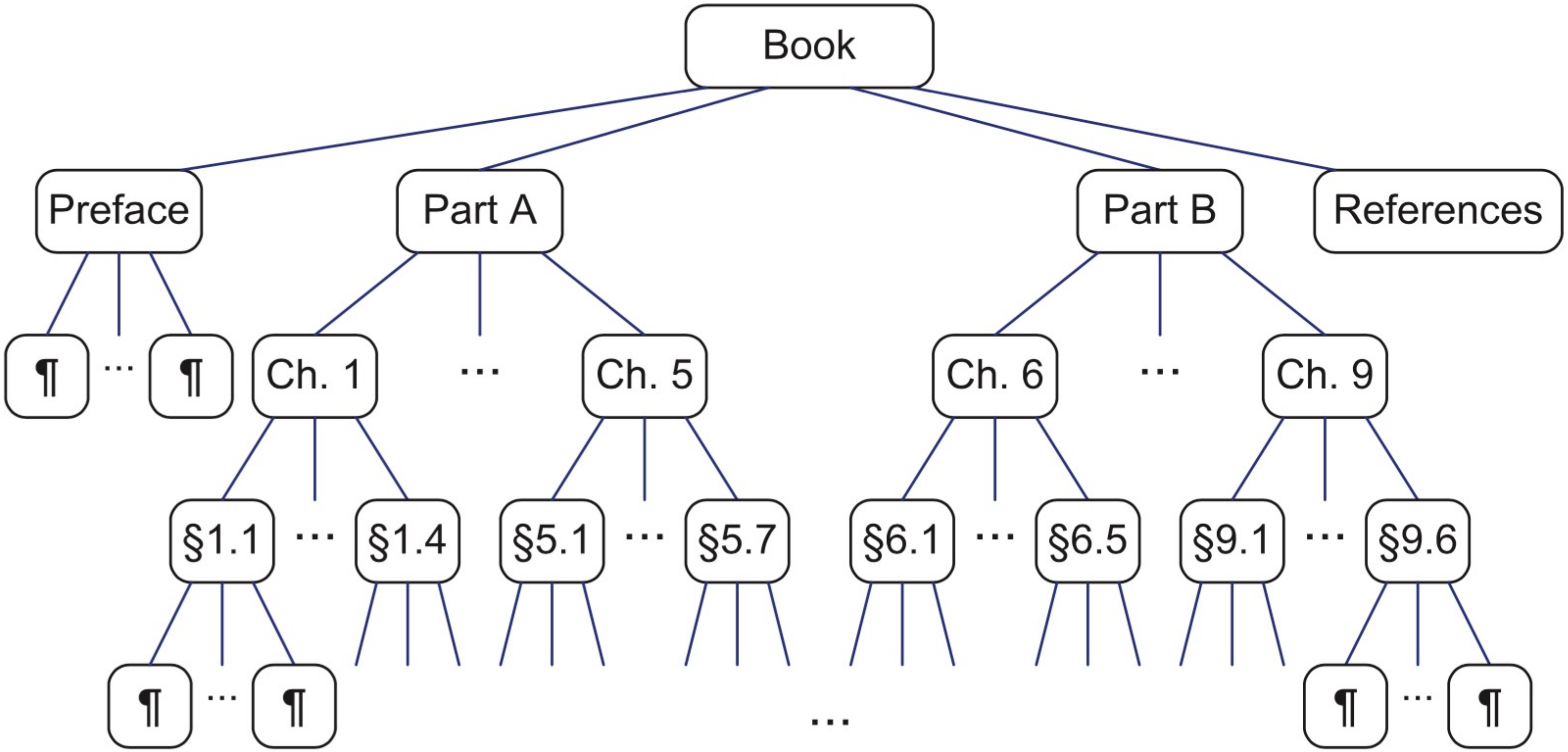
Formal Definition of Tree

- we define tree **T** to be a **set of nodes** storing elements in a parent-child relationship with the following properties:
  - 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!
- So, with the above definition we can define it recursively:
  - a tree **T** is either empty or consists of a node **r**, called the root of **T**, and a (possibly empty) set of trees whose roots are the children of **r**



Ordered Tree Examples

- Ordered Tree: A tree is ordered if there is a linear ordering defined for the children of each node
  - linear order relationship existing between siblings
- Book Organization Tree



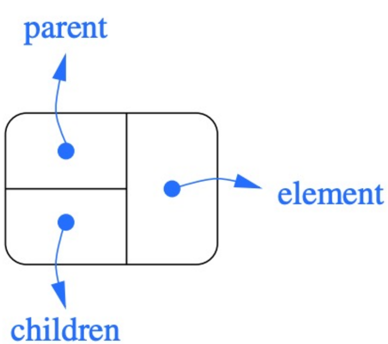
Tree ADT

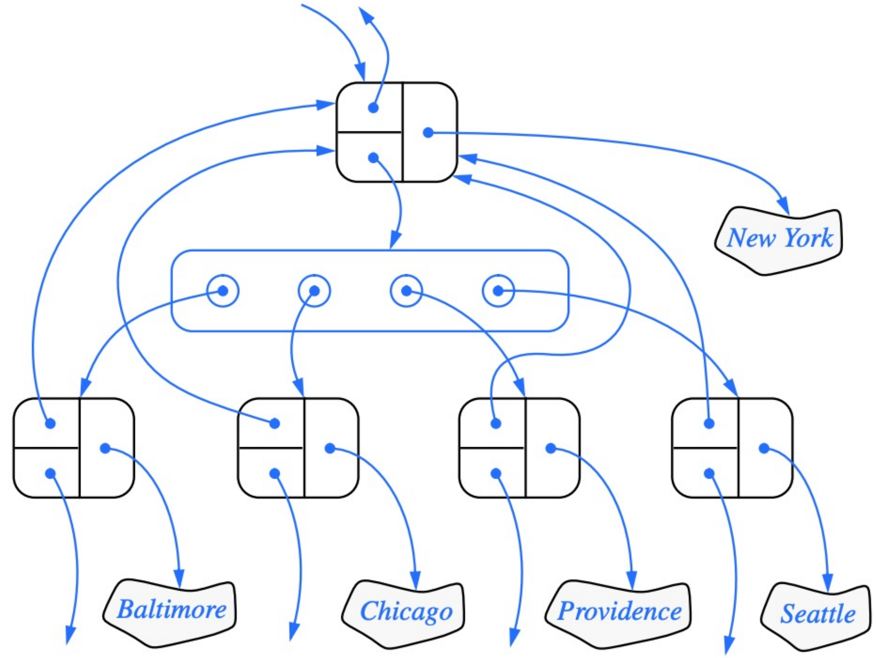
- We use **positions** to abstract nodes
- Generic methods:
  - integer **size**()
  - boolean **empty**()
- Accessor methods:
  - position **root**()
  - list<position> **positions**()
- Position-based methods:
  - position p.**parent**()
  - list<position> p.**children**()
- Query methods:
  - boolean p.**isRoot**()
  - boolean p.**isExternal**()
- Additional update methods may be defined by data structures implementing the Tree ADT

Tree ADT

- General Tree Implementation

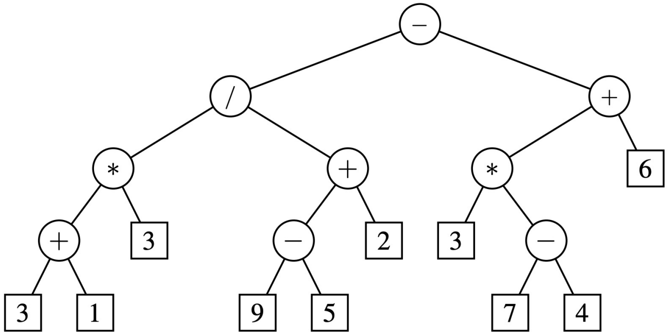
```
template <typename E>                    // base element type
class Position<E> {                      // a node position
public:
  E& operator*();                        // get element
  Position parent() const;               // get parent
  PositionList children() const;         // get node's children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





Tree Traversals

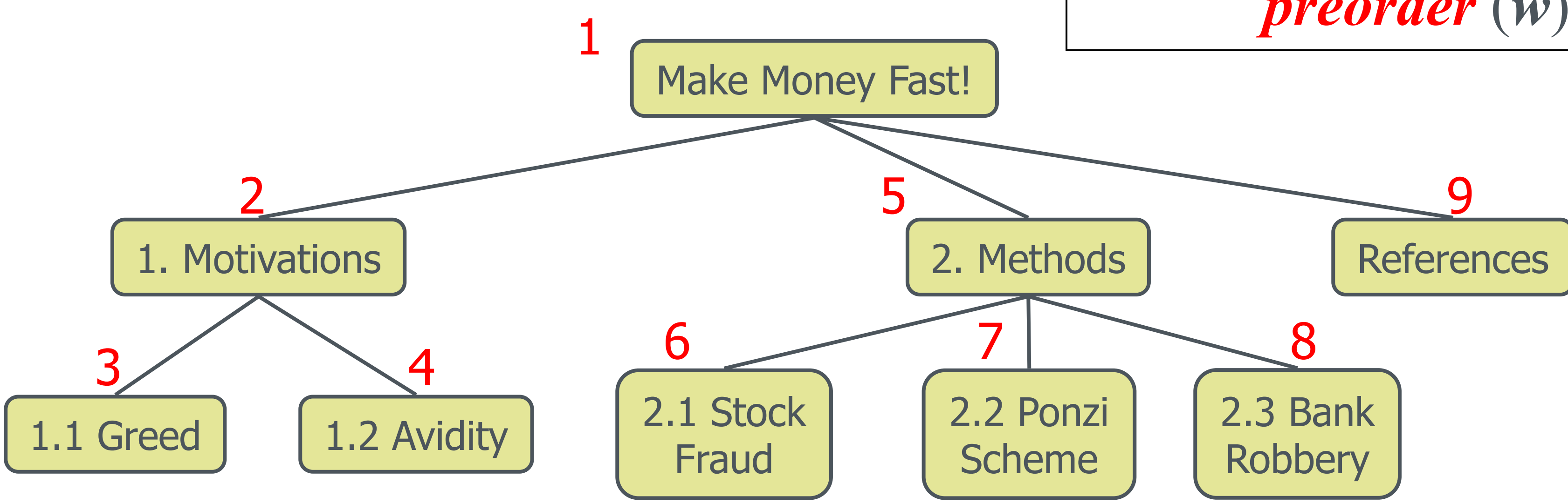
- We were able to easily traverse a linear data structure
  - The traversal of a tree is not trivial
- A **traversal** of a tree **T** is a systematic way of accessing or visiting all the nodes of T.
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal
  - Breadth-first traversal



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

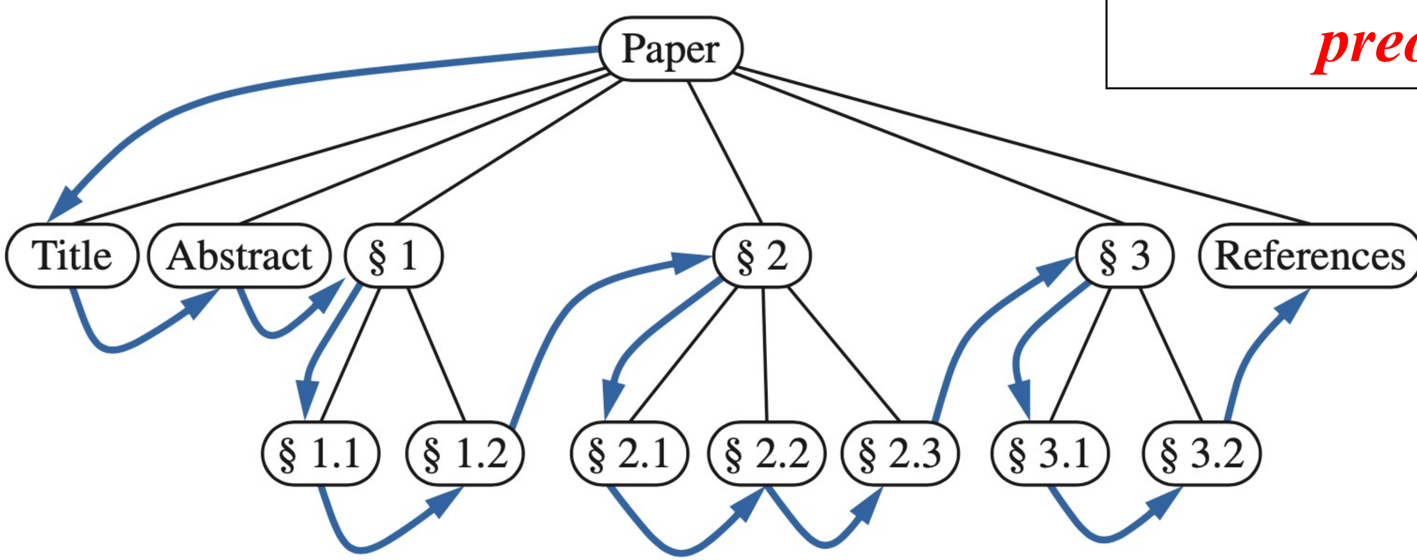
Algorithm ***preOrder*****(*v*)**
    visit**(*v*)**
    for each child ***w*** of ***v***
        preorder **(*w*)**



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm *preOrder*(v)
    visit(v)
    for each child w of v
        preorder (w)



Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

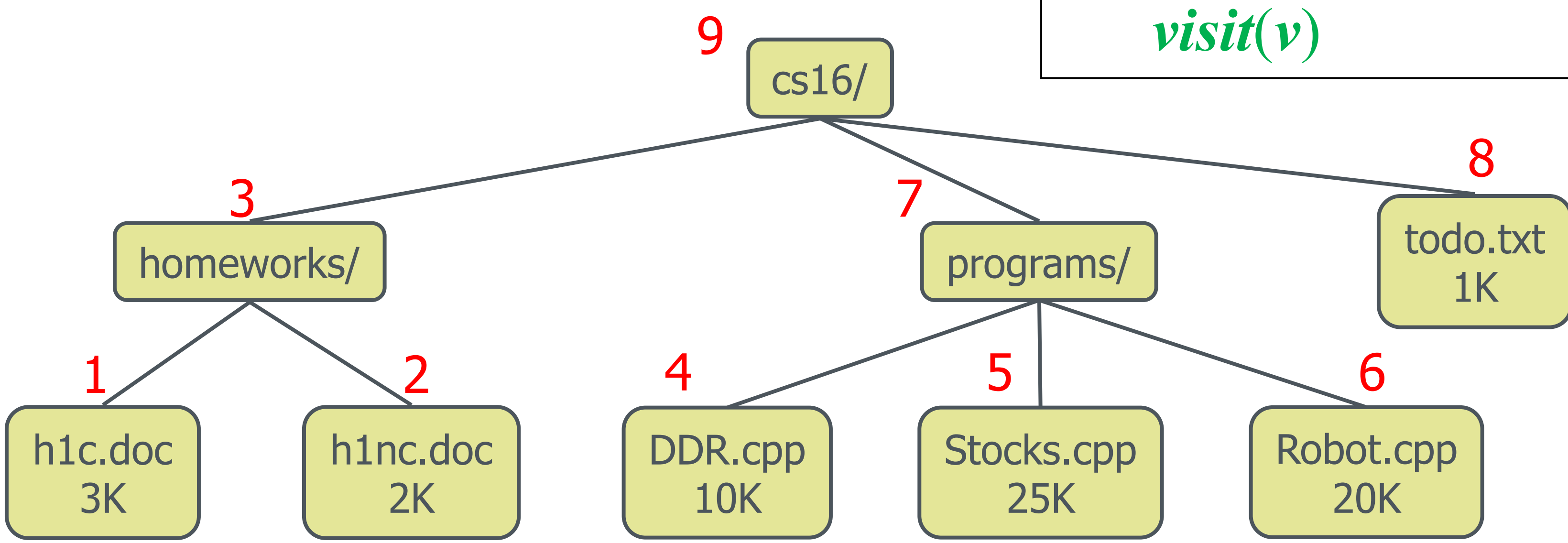
Algorithm ***preOrder***(v)
    visit(v)
    for each child w of v
        preorder (w)

```
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

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

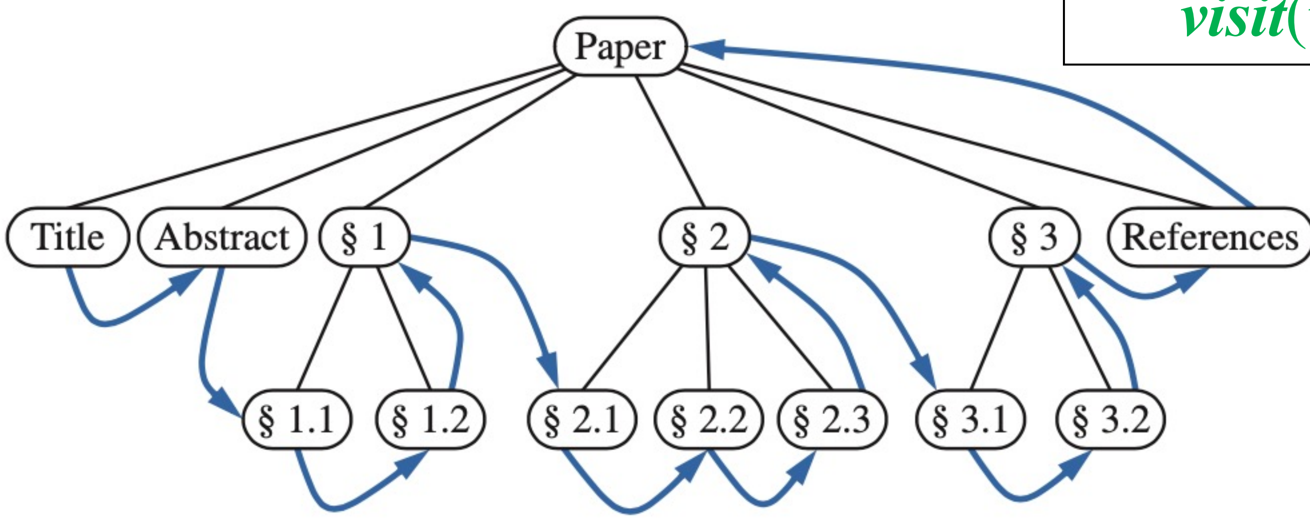
Algorithm ***postOrder*****(*****v*****)**
  for each child ***w*** of ***v***
    postorder **(*****w*****)**
  visit**(*****v*****)**



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm ***postOrder(v)***
 for each child *w* of *v*
  postorder (*w*)
 visit(v)



Postorder Traversal

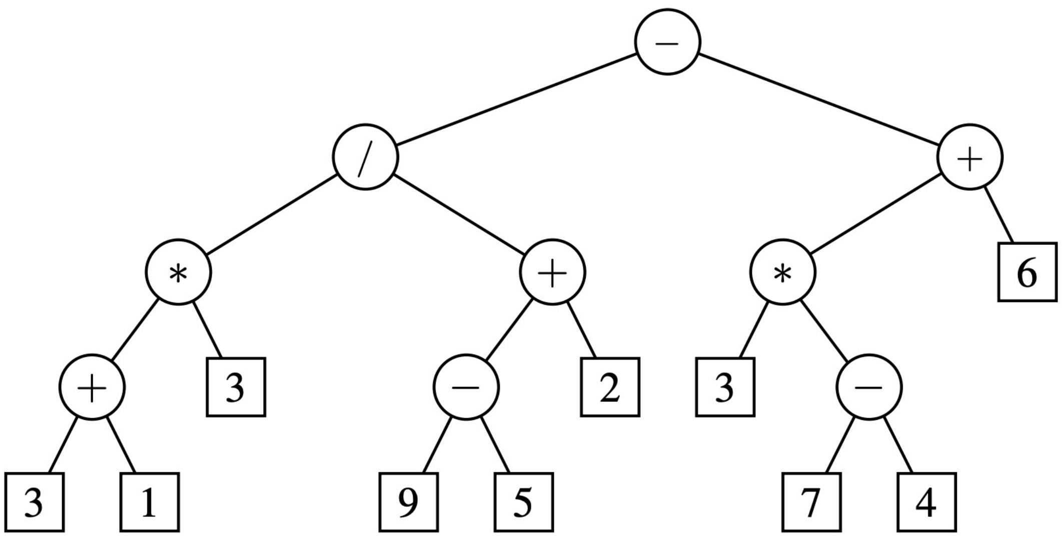
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm ***postOrder(v)***
    for each child ***w*** of ***v***
        postorder (***w***)
    visit(v)

```
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

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories



Algorithm ***postOrder*(*v*)**

  for each child ***w*** of ***v***

    postorder (***w***)

  ***visit*(*v*)**

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

Postorder?

Questions?