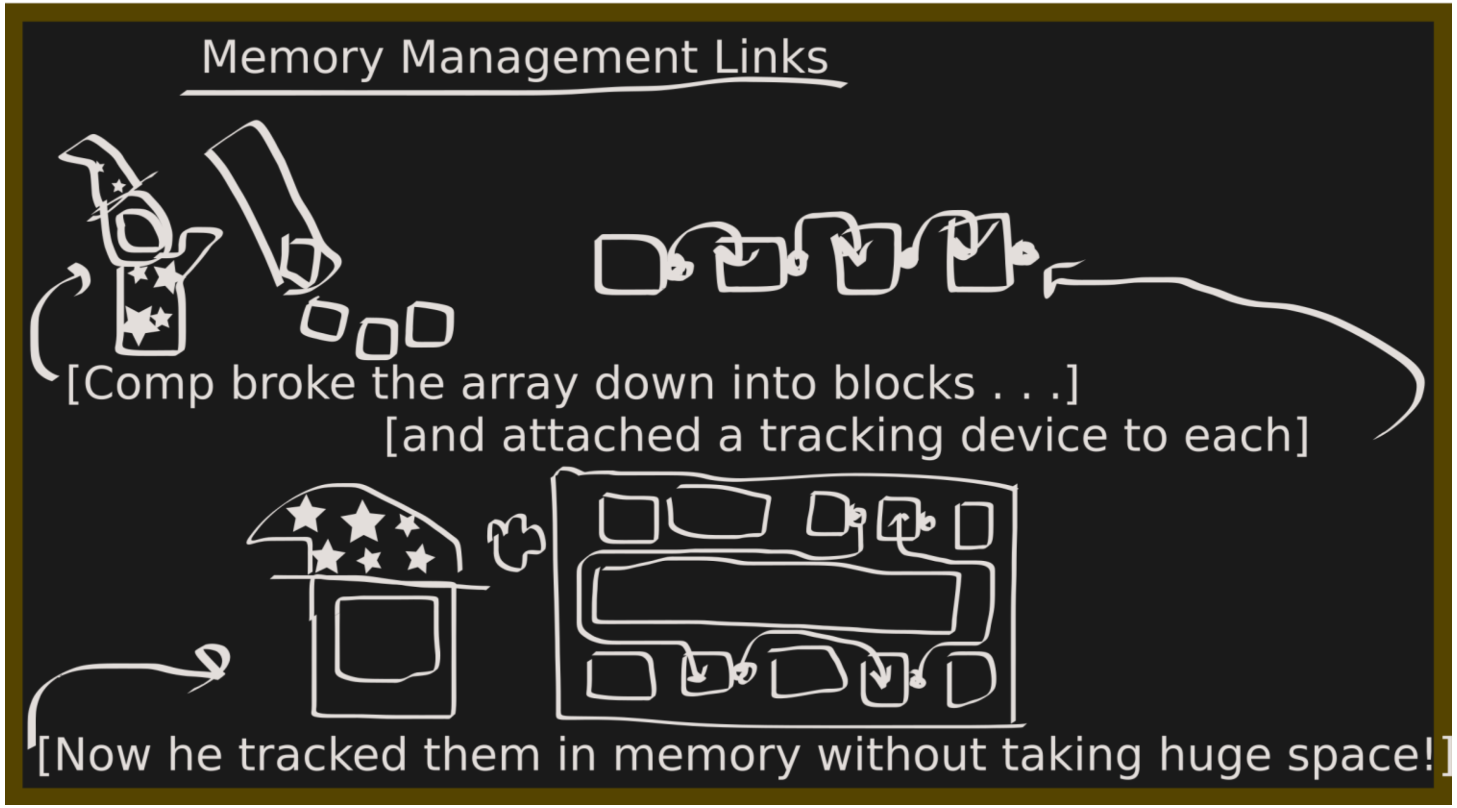
TREES

School of Artificial Intelligence

PREVIOUSLY ON DS&A

- LinkedList （链表）
- Singly Linked List （单链表）
- Doubly Linked List （双链表）
- Positional List
- Linked Based Sequence vs. Array Based Sequence



THE POSITIONAL LIST ADT

- Position ADT
  - P.element(): return the element stored at position p
- Positional List ADT
  - L.first(): first element of L, or None if L is empty
  - L.last(): last element of L, or None if L is empty
  - L.before(p): the position in L immediately before p, or None if p is the first position
  - L.after(p): the position in L immediately after p, or None if p is the last position
  - L.is_empty(): true if L is empty
  - len(L): number of elements in the list
  - iter(L): returns a forward iterator for the elements of the list

THE POSITIONAL LIST ADT

- Positional List ADT
    - L.add_first(e): insert a new element e at the front of L, return the position of the new element
    - L.add_last(e): insert a new element e at the back of L, return the position of the new element
    - L.add_before(p, e): insert a new element e before position p in L, return the position of the new element
    - L.add_after(p, e): insert a new element e after position p in L, return the position of the new element
    - L.replace(p, e): replace the element at position p with element e, returning the element formerly at position p
    - L.delete(p): remove and return the element at position p in L

DOUBLY LINKED LIST IMPLEMENTATION

- Validating a position
  - Necessary to avoid corrupting the list
  - P needs to be of type Position
  - The container of p is the linkedlist
  - Position is not valid if the node is not in the

```
#---------------------------- utility method ----------------------------
def _validate(self, p):
  """Return position's node, or raise appropriate error if invalid."""
  if not isinstance(p, self.Position):
    raise TypeError('p must be proper Position type')
  if p._container is not self:
    raise ValueError('p does not belong to this container')
  if p._node._next is None:                  # convention for deprecated nodes
    raise ValueError('p is no longer valid')
  return p._node

#---------------------------- utility method ----------------------------
def _make_position(self, node):
  """Return Position instance for given node (or None if sentinel)."""
  if node is self._header or node is self._trailer:
    return None                              # boundary violation
  else:
    return self.Position(self, node)         # legitimate position
```

LINKED-BASED VS. ARRAY BASED SEQUENCES

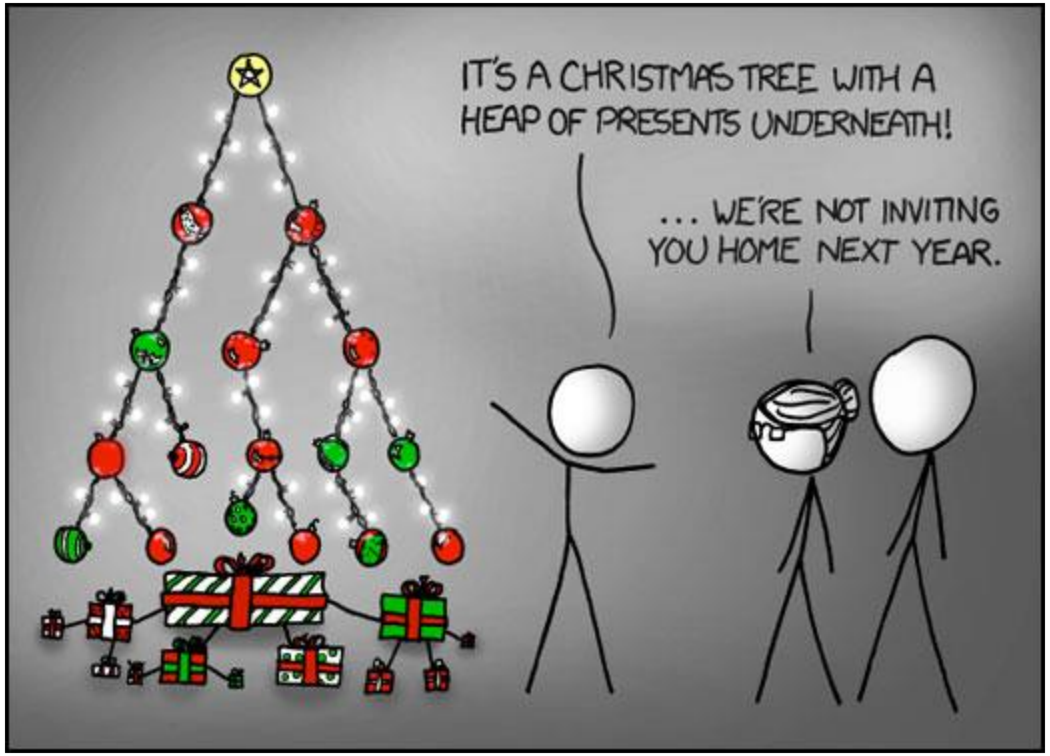
- Advantages of array-based sequences
  - O(1) time to access an element based on an integer index
    - Getting the k^{th} element in a linkedlist takes O(k) time, or O(n-k) time
  - Operations with equivalent asymptotic bounds run more efficient with an array based structure
    - enqueue() operation for array: compute new index, increment of an integer, storing a reference to the element in the array
    - enqueue() operation for linkedlist: creating a new node, linking the nodes, increment of an integer
    - O(1) time in either model, but more CPU time needed for linked list
  - Less memory than linked list
    - Linkedlist uses more auxiliary memory: reference to element, reference to next/prev

LINKED-BASED VS. ARRAY BASED SEQUENCES

- Advantages of link-based sequences
    - Worst-case time bounds can be obtained for operations in a linked list
        - Amortised bounds can be obtained by array-based sequence
        - But remember sequential applications vs. real-time applications
    - O(1) time insertions and deletions
        - It is easy to insert and delete elements at any position in a linkedlist
        - pop() with index k needs O(n-k+1) time, because of the shifting of elements

THIS LECTURE

- Trees
- Definition of a Tree
- Binary Trees

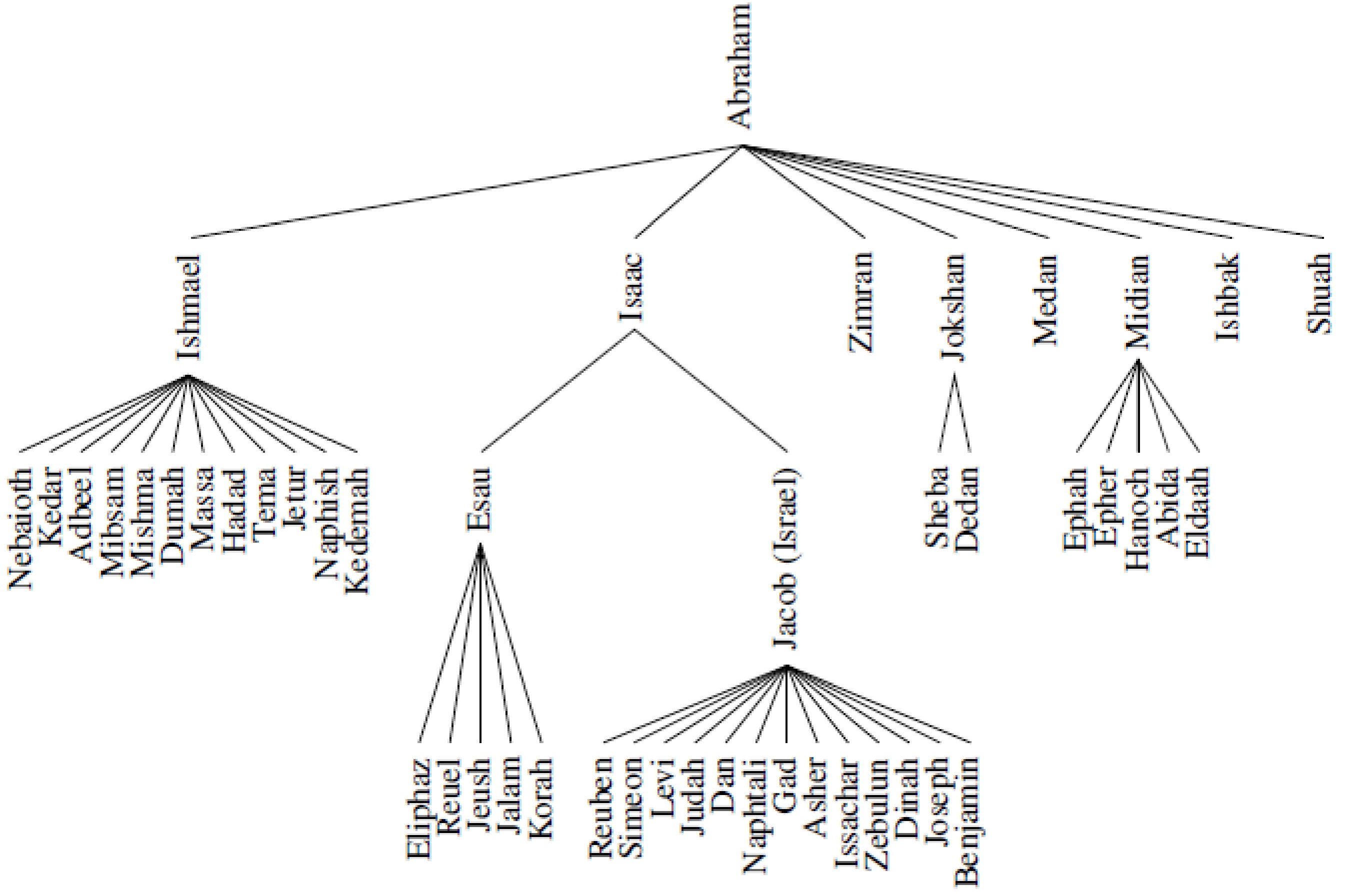


GENERAL TREES

- Tree: most important non-linear data structure in computing
- Why trees
  - some algorithms run much faster on trees than on linear data structures such as array-based lists or linked lists
  - Natural organisation for data - file systems, graphical user interfaces, databases, web sites
- Non-linear: an organizational relationship that is richer than the simple “before” and “after relationship between objects in sequences
- Trees are hierarchical, with some objects being “above” and some “below” others

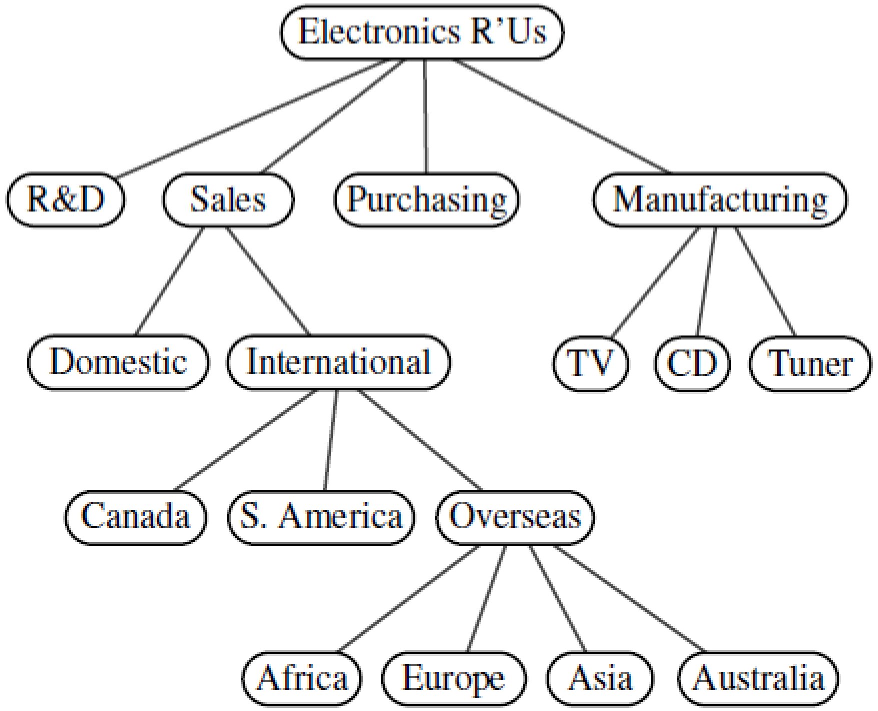
GENERAL TREES

- The terminology for *Tree* mostly originates from "family trees"
- The notion of "parent", "child", "ancestor" and "descendant"



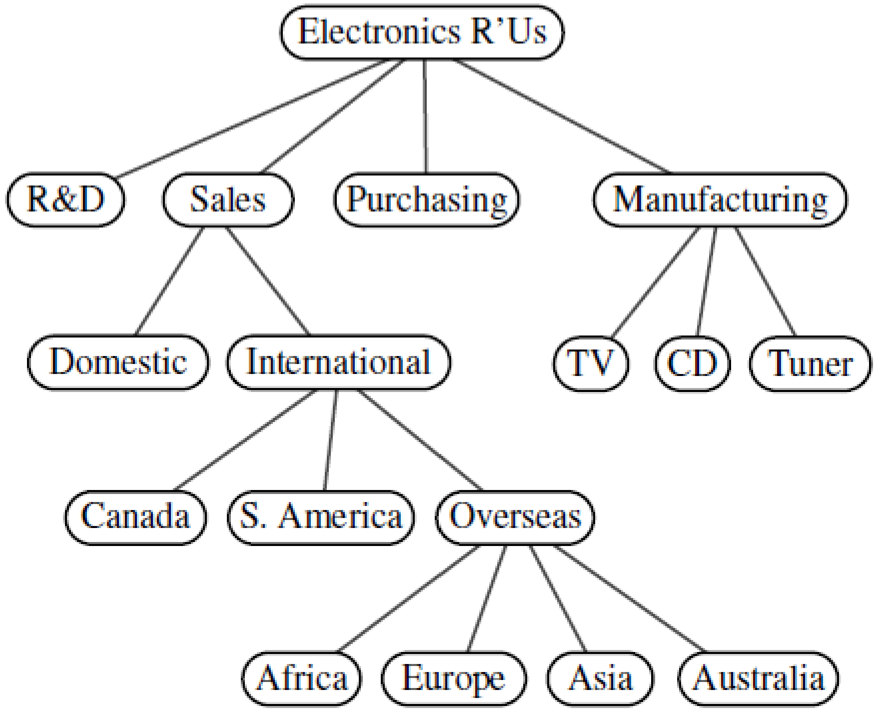
GENERAL TREES

- Definitions and properties
- Tree: abstract data type that stores elements hierarchically
- Node: a container that contains an element stored in the tree
- Parent node: a node directly connected and above
- Child node: a node directly connected and below
- Root: top level element



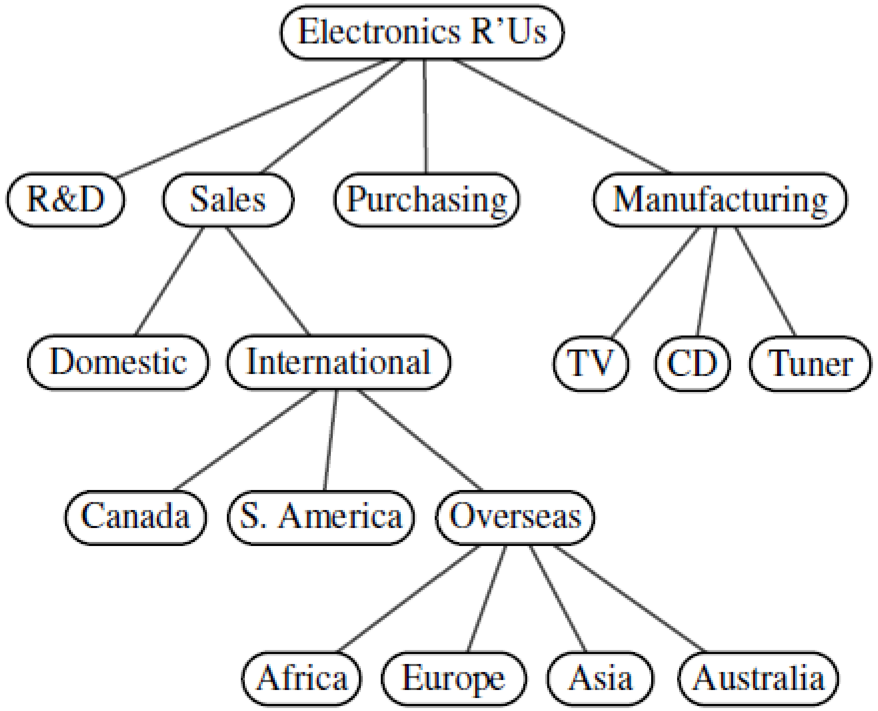
GENERAL TREES

- A tree T as a set of nodes storing elements such that the nodes have a parent-child relationships that satisfies the following properties:
  - If T is non-empty, 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
- Empty tree: no nodes at all
- “a tree T is either empty or consists of a node r, called the root of T, and a set of subtrees whose roots are the children of r”



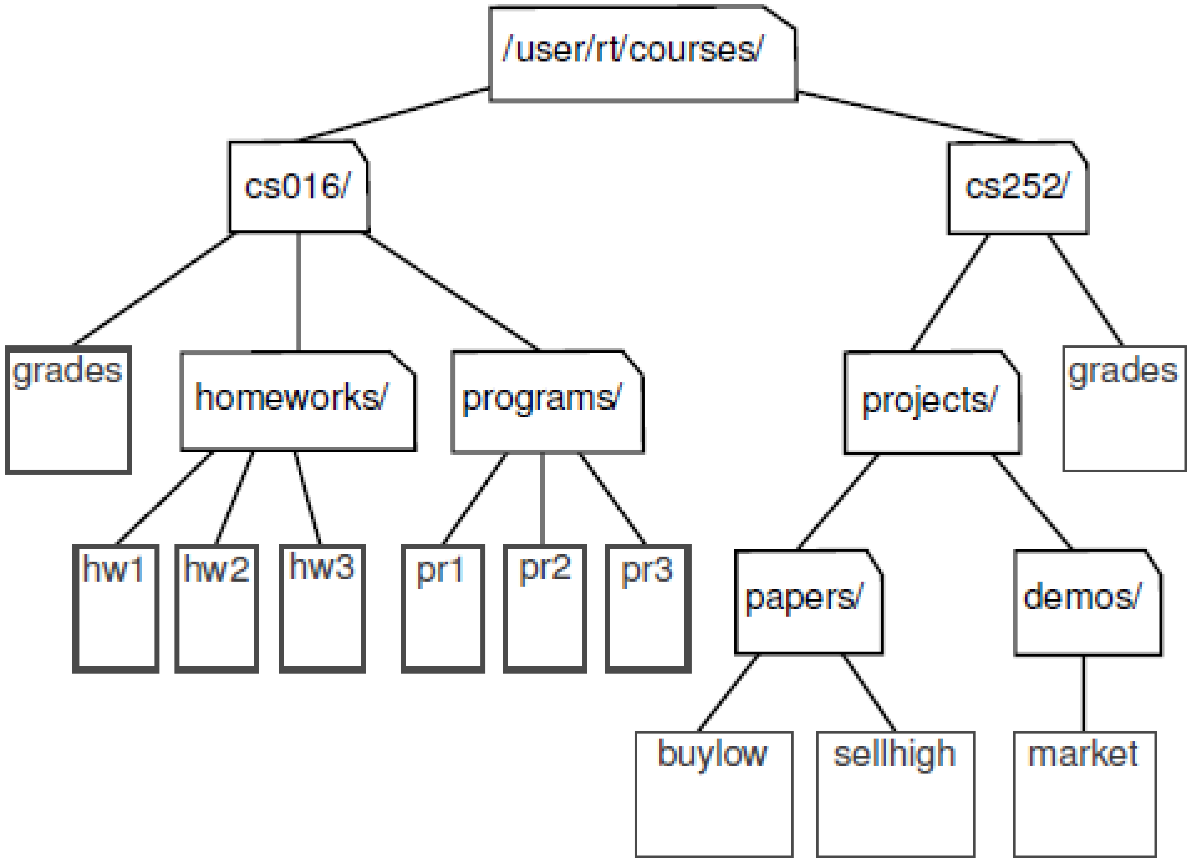
GENERAL TREES

- Siblings(兄弟结点): two nodes that are children of the same parent
- External node（外部结点）: if node v has no children
- Internal node（内部结点）: if node v has one or more children
- External nodes: a.k.a. **leaves** （叶结点）



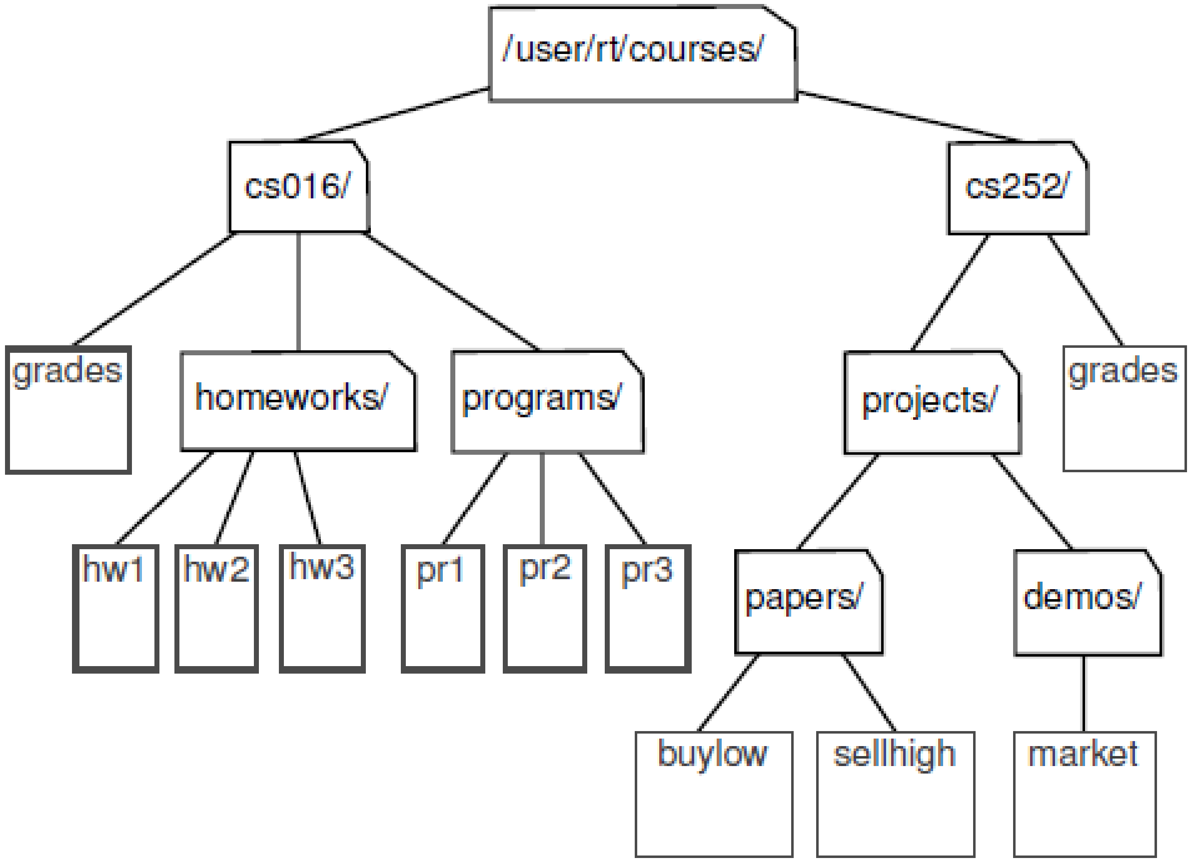
GENERAL TREES

- Internal nodes: folders
- Leaves: files
- Root: /user/rt/courses
    - This is a sub-tree, what's the "root" of the entire tree?



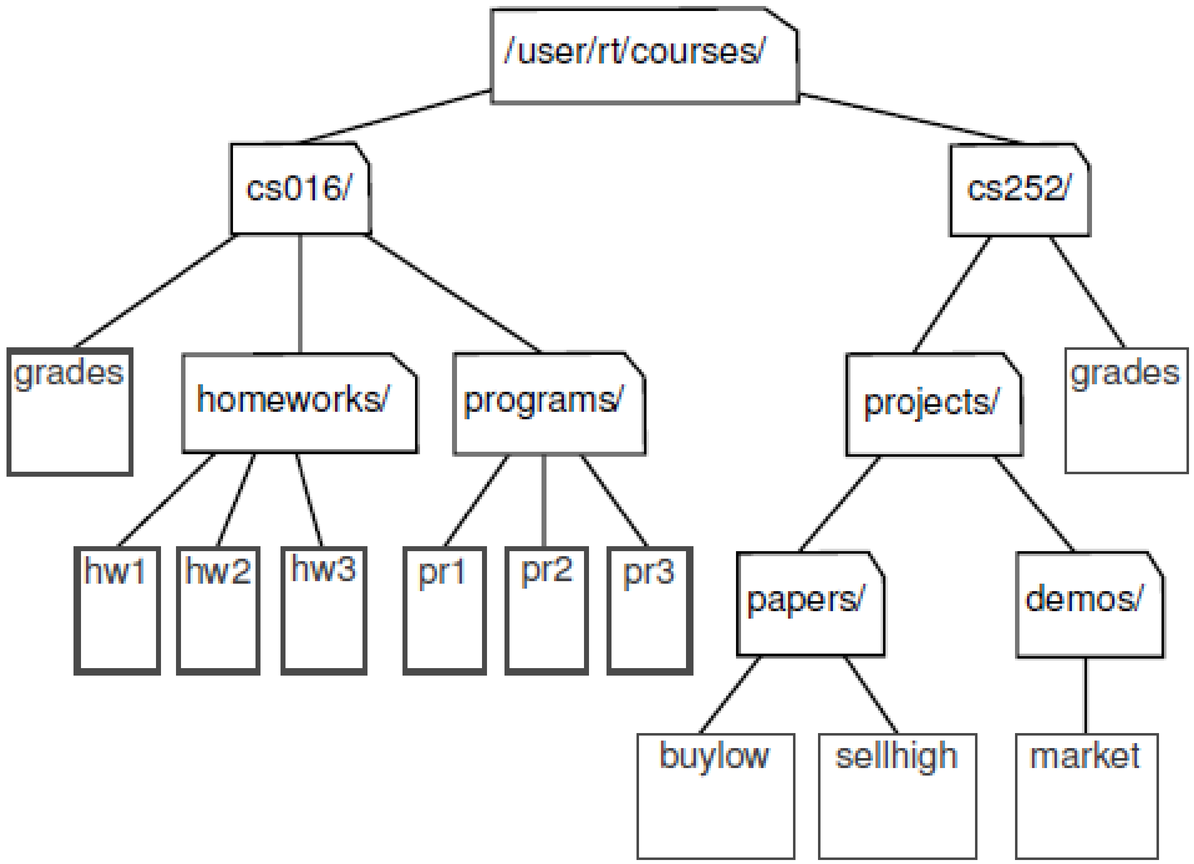
GENERAL TREES

- Ancestor(祖先结点): a node u is an ancestor of v, if 1)u = v; 2)u is an ancestor of the parent of v
- Descendant (子孙结点): v is a descendant of a node u if u is an ancestor of v
- cs252/ is an ancestor of /papers
- pr3 is a descendant of cs016/



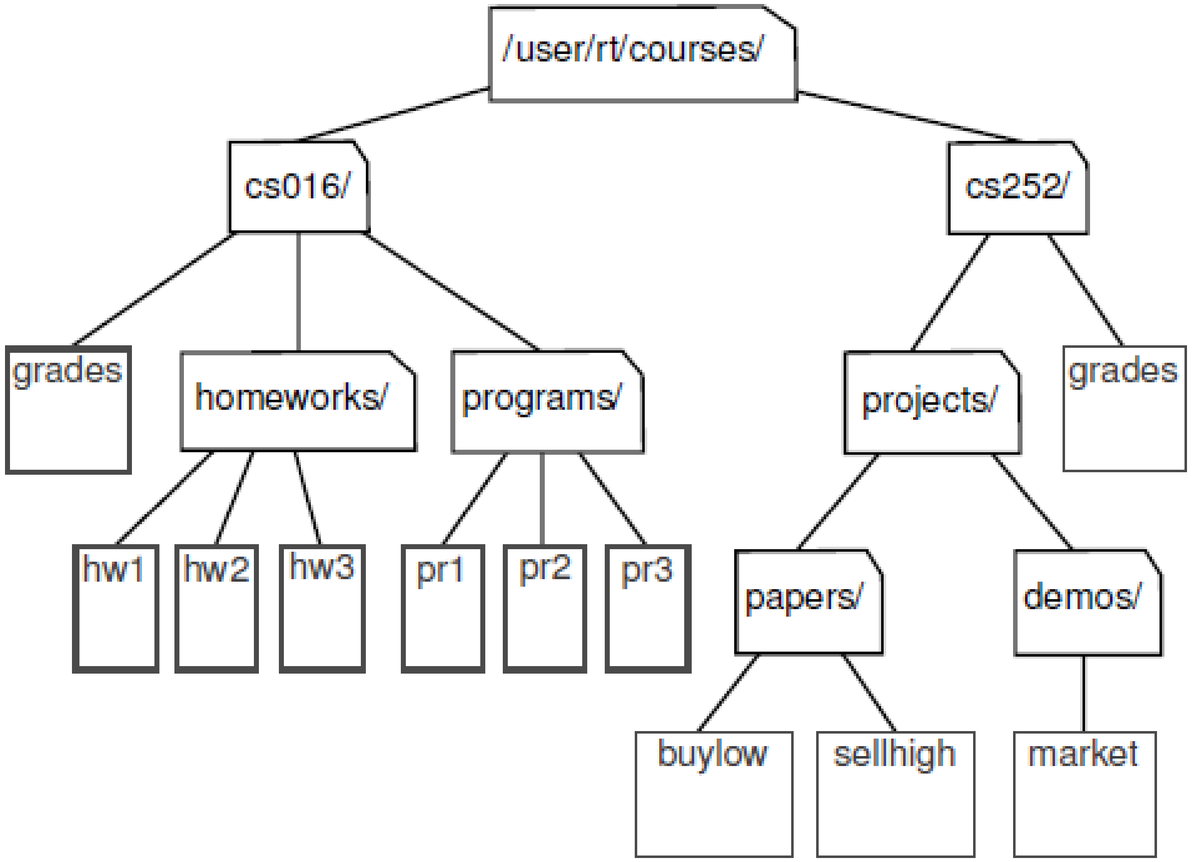
GENERAL TREES

- Subtree of T rooted at a node v is the tree consisting of all the descendants of v in T (including v)
- Subtree rooted at cs016/
  - Cs016/
  - Grades/
  - Homeworks/
  - Programs/
  - Hw1, hw2, hw3
  - Pr1, pr2, pr3



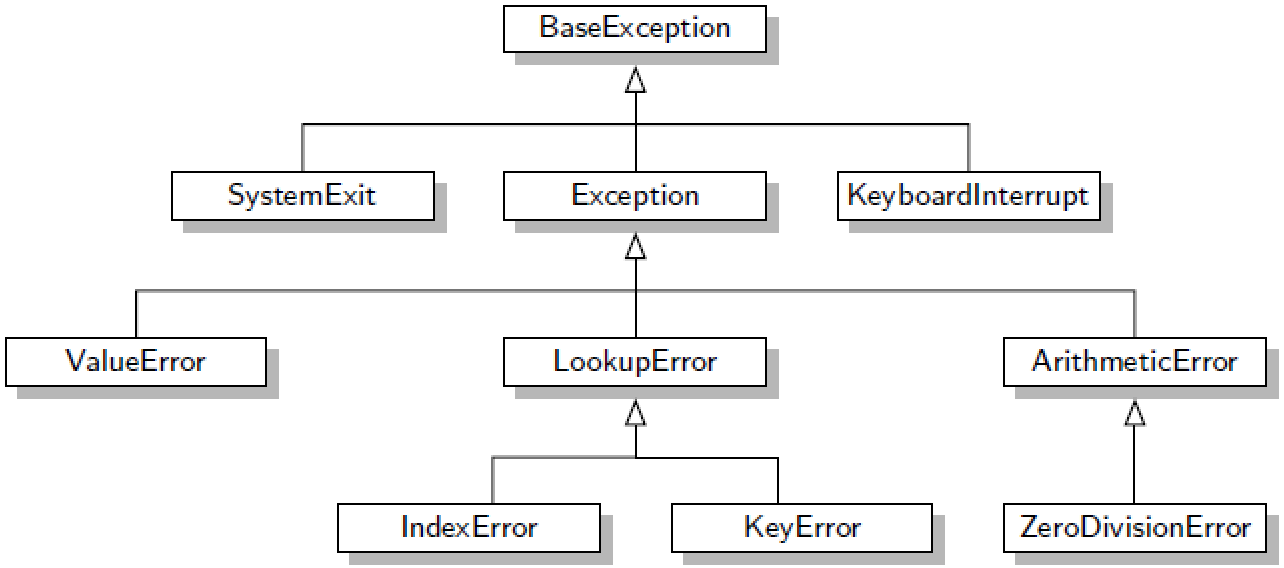
EDGES AND PATHS

- Edge: pair of nodes (u,v) such that u is the parent of v, or vice versa.
- Path: sequence of nodes such that any two consecutive nodes in the sequence form an edge
- E.g.: cs252/projects/demos/market



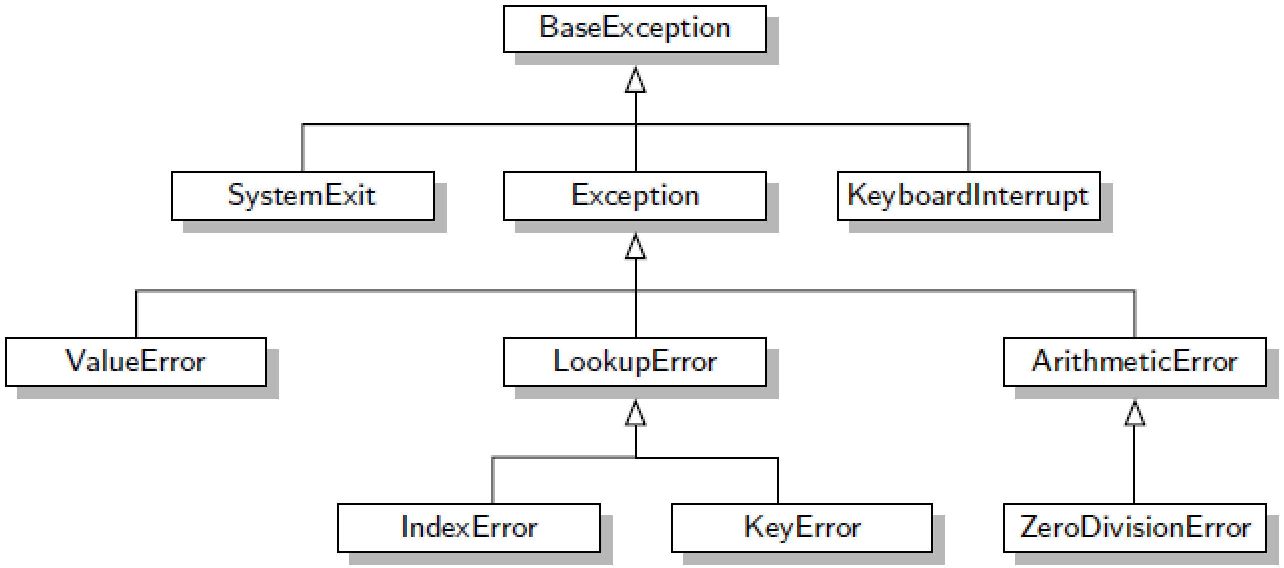
PYTHON'S INHERITANCE TREE

- Inheritance relation between classes in the Python language
- Root?
- Internal Nodes?
- External Nodes?
- Siblings?
- Ancestors?
- Descendants?



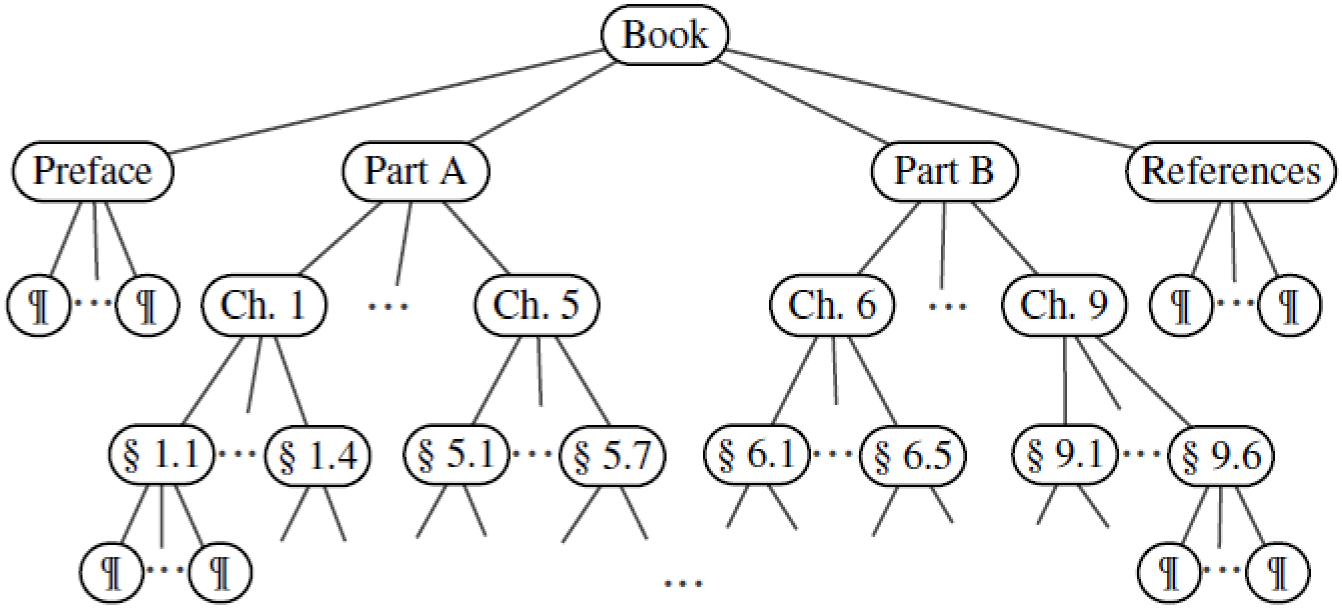
PYTHON'S INHERITANCE TREE

- But this is just a sub-tree
- In Python, there is an overall base class called object
- All other classes inherit object
- In other words, object is the ancestor of all other classes, including the ones that you create



ORDERED VS. UNORDERED

- A tree is ordered if there is a linear order among the children of each node
- Book is an ordered tree
  - Internal nodes: parts, chapters, sections, paragraphs, tables, figures, etc.
  - Root of the tree: book
- Examples of ordered trees?
- Examples of unordered trees?



THE TREE ADT

- Remember Positional list?
  - Operations on a positional list returns a position, which points to a location in the list that contains an element
  - We will do the same for tree
- Position ADT
  - p.element(): return the element stored at position p
- Tree ADT
  - T.root(): return the position of the root
  - T.is_root(p): returns tree if position p is the root of tree T
  - T.parent(p): returns the position of the parent of position p, or None if p is the root of T
  - T.num_children(p): returns the number of children of position p

THE TREE ADT

- Tree ADT
  - T.children(p): returns a collection of the children of position p
  - T.is_leaf(p): returns true if position p does not have any children (i.e. it is a leaf node)
  - len(T): returns the number of elements in T
  - T.is_empty(): returns true if T does not contain any elements
  - T.positions(): returns a collection of all positions of tree T
  - Iter(T): returns a iterator of all elements stored in T
- All above functions should raise an error if position p is not in T (invalid)

THE TREE ADT

- Some remarks
- For ordered trees: T.children(p) returns the children in the natural order (leftmost sibling node to rightmost sibling node)
- If p is a leaf, then T.children(p) returns an empty collection
- If T is empty, then T.positions() and iter(T) returns an empty collection/iterator
- Mutating methods: discussed later when we implement trees

OBJECT ORIENTED PROGRAMMING

- An abstract base class for Tree
- Define default behaviours for operations
- Greater re-use
- Position class:
    - Element() and __eq__() not implemented
    - NotImplementedError when called
    - Typical for Object Oriented Programming
    - You need to define your own subclass for Position that overrides these two

```python
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- nested Position class -------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedError('must be implemented by subclass')

    def __eq__(self, other):
      """Return True if other Position represents the same location."""
      raise NotImplementedError('must be implemented by subclass')

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)            # opposite of __eq__
```

OBJECT ORIENTED PROGRAMMING

- Tree abstract base class
- root(), parent(), num_children(), children(), __len__() not implemented

```
# ---------- abstract methods that concrete subclass must support ----------
def root(self):
  """Return Position representing the tree's root (or None if empty)."""
  raise NotImplementedError('must be implemented by subclass')

def parent(self, p):
  """Return Position representing p's parent (or None if p is root)."""
  raise NotImplementedError('must be implemented by subclass')

def num_children(self, p):
  """Return the number of children that Position p has."""
  raise NotImplementedError('must be implemented by subclass')

def children(self, p):
  """Generate an iteration of Positions representing p's children."""
  raise NotImplementedError('must be implemented by subclass')

def __len__(self):
  """Return the total number of elements in the tree."""
  raise NotImplementedError('must be implemented by subclass')
```

OBJECT ORIENTED PROGRAMMING

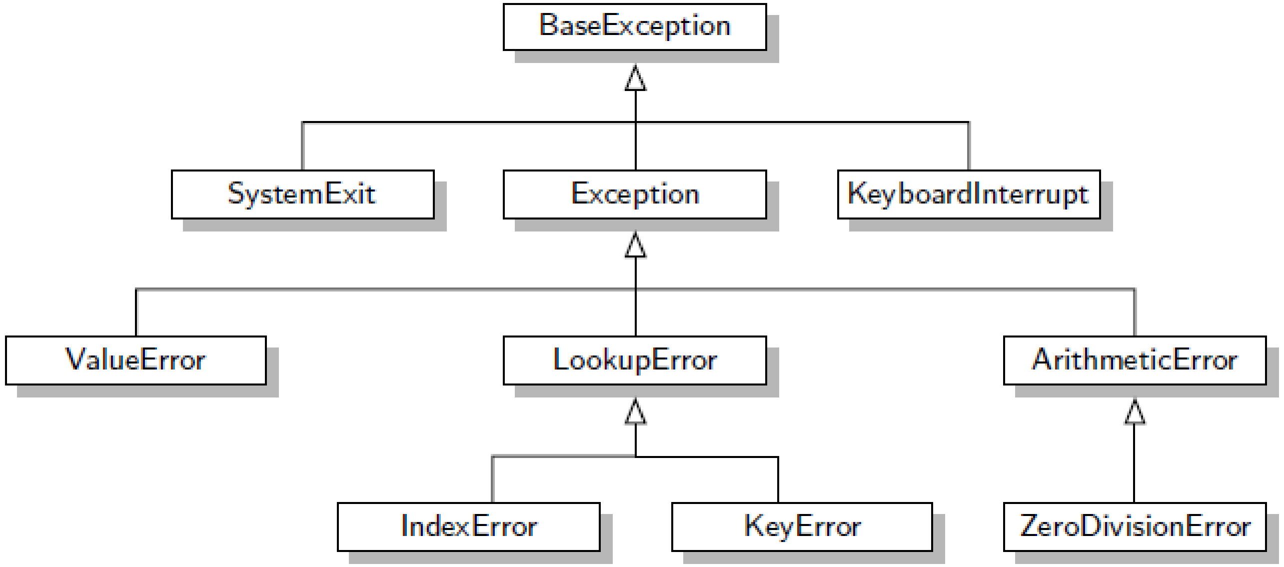
- Tree abstract base class
- Some functions are implemented as concrete methods
  - Why?
- Is it sensible to create a direct instance （实例） of the abstract Tree class?

```
# ---------- concrete methods implemented in this class ----------
def is_root(self, p):
  """Return True if Position p represents the root of the tree."""
  return self.root( ) == p

def is_leaf(self, p):
  """Return True if Position p does not have any children."""
  return self.num_children(p) == 0

def is_empty(self):
  """Return True if the tree is empty."""
  return len(self) == 0
```

Code Fragment 8.2: Some concrete methods of our Tree abstract base class.

COMPUTING DEPTH

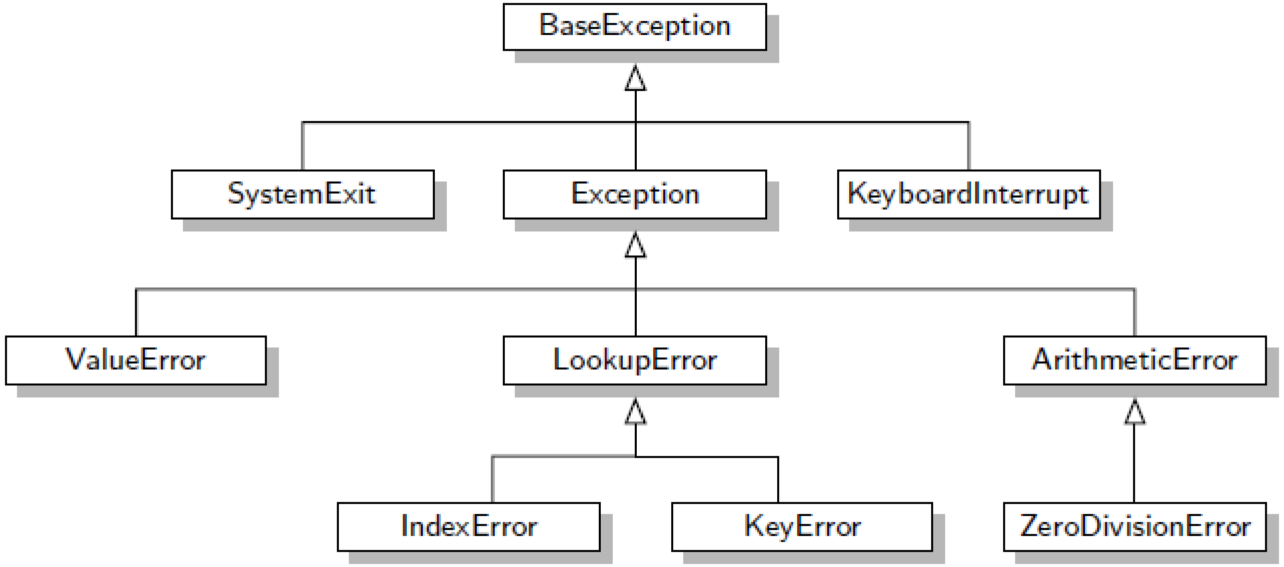
- Tree T, position of a node p
- The depth of p: number of ancestors of p, excluding p itself.
- Depth of LookUpeError: 2
- Formal definition:
  - If p is the root then depth of p is 0
  - Otherwise, the depth of p is one plus the depth of the parent of p




```
def depth(self, p):
  """Return the number of levels separating Position p from the root."""
  if self.is_root(p):
    return 0
  else:
    return 1 + self.depth(self.parent(p))
```

COMPUTING DEPTH

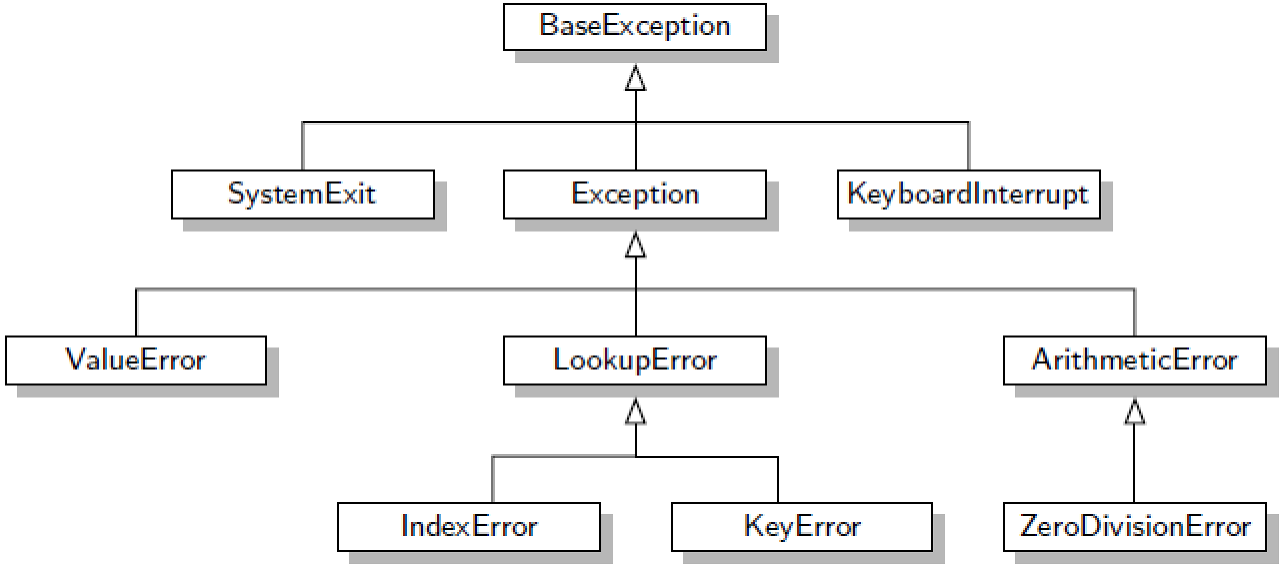
- Running time T.depth(p)?
- O(d + 1): d = depth of p
- Worst case O(n), n = total number of position of T




```
def depth(self, p):
  """Return the number of levels separating Position p from the root."""
  if self.is_root(p):
    return 0
  else:
    return 1 + self.depth(self.parent(p))
```

COMPUTING HEIGHT

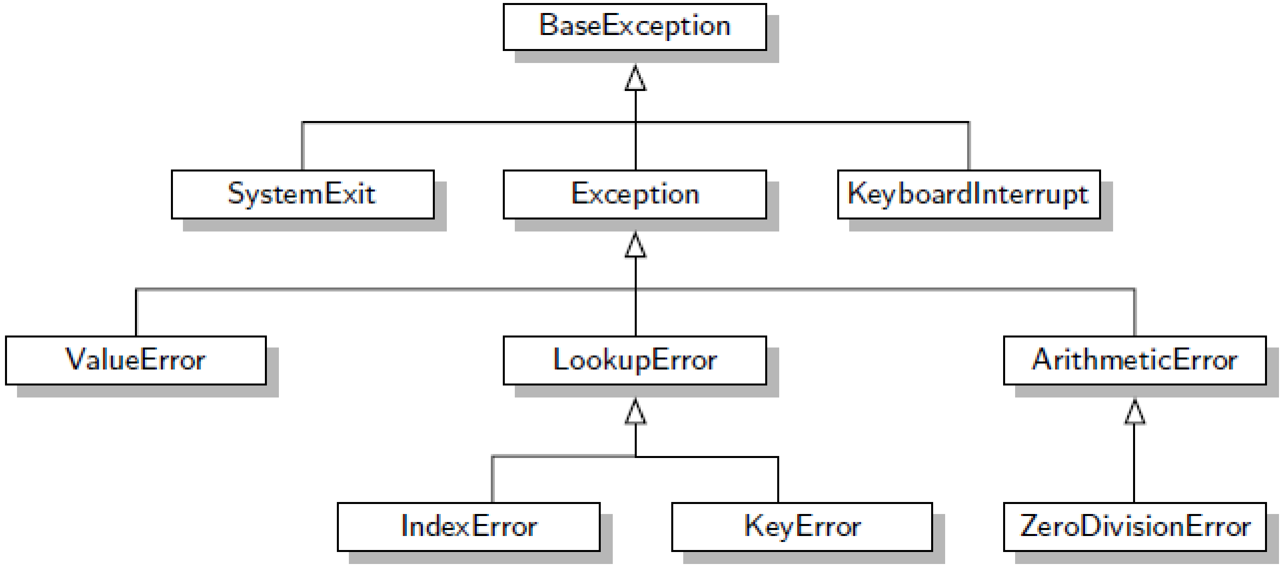
- Height of a position p in tree T:
  - If p is a leaf, its height is 0
  - Otherwise, the height of p is one more than the maximum of the heights of p's children
- Height of a nonempty T is the height of the root of T
- The Exception tree has a height 3
- The height of a non-empty tree = the maximum of the depths of its leaf positions




```
def _height1(self):                      # works, but O(n^2) worst-case time
  """Return the height of the tree."""
  return max(self.depth(p) for p in self.positions( ) if self.is_leaf(p))
```

COMPUTING HEIGHT

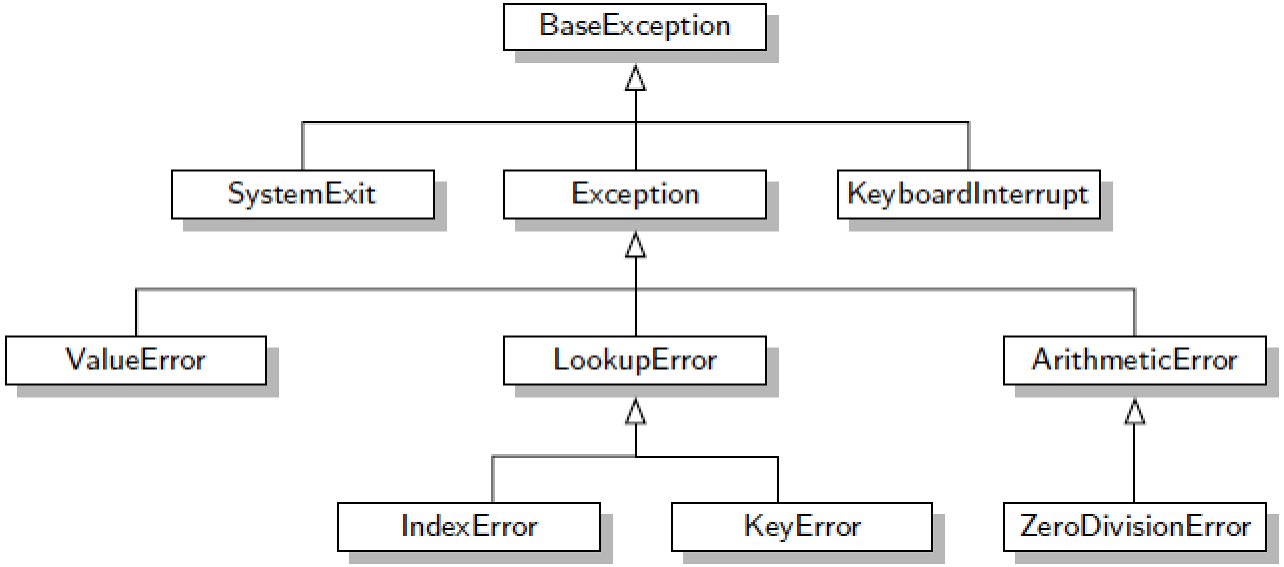
- Complexity of height1()?
- positions(): can run O(n)
- Depth(p): we have established that this can run in O(n)
- Height1(): $O(n^2)$




```
def _height1(self):                # works, but O(n^2) worst-case time
  """Return the height of the tree."""
  return max(self.depth(p) for p in self.positions( ) if self.is_leaf(p))
```

COMPUTING HEIGHT

- An improved version with recursion
- Complexity?



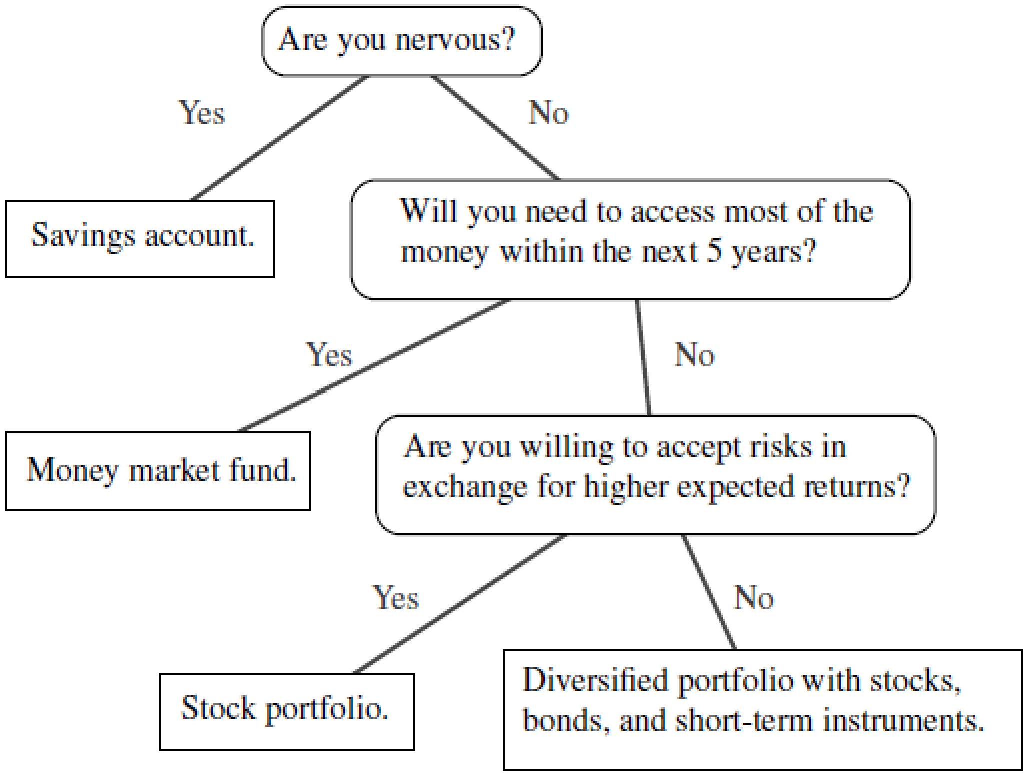

```
def _height2(self, p):                    # time is linear in size of subtree
  """Return the height of the subtree rooted at Position p."""
  if self.is_leaf(p):
    return 0
  else:
    return 1 + max(self._height2(c) for c in self.children(p))
```

BINARY TREES （二叉树）

- Binary tree: ordered tree such that
  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 order of children of a node
- Subtrees: left subtree and right sub tree
- A binary tree is **proper** (标准的/完全) if each node has either zero or two children
  - Also full binary trees

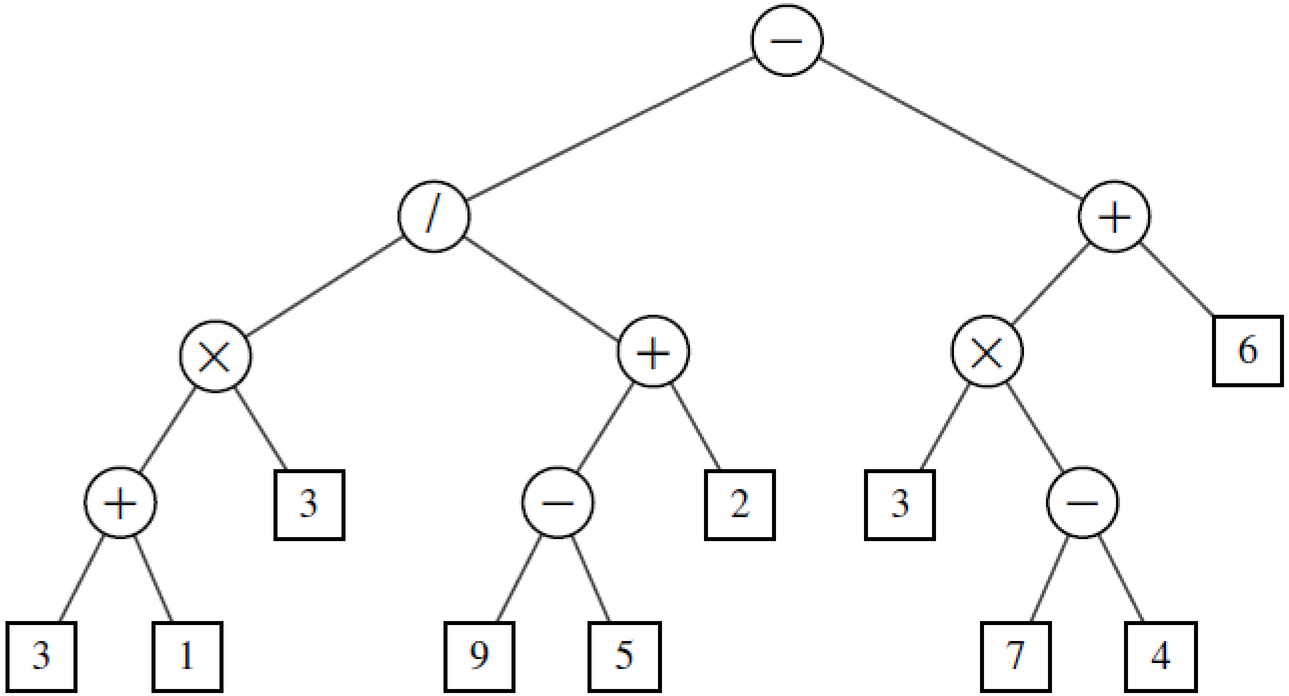
BINARY TREES

- Example: decision trees
- A number of different outcomes based on answers to yes-no questions



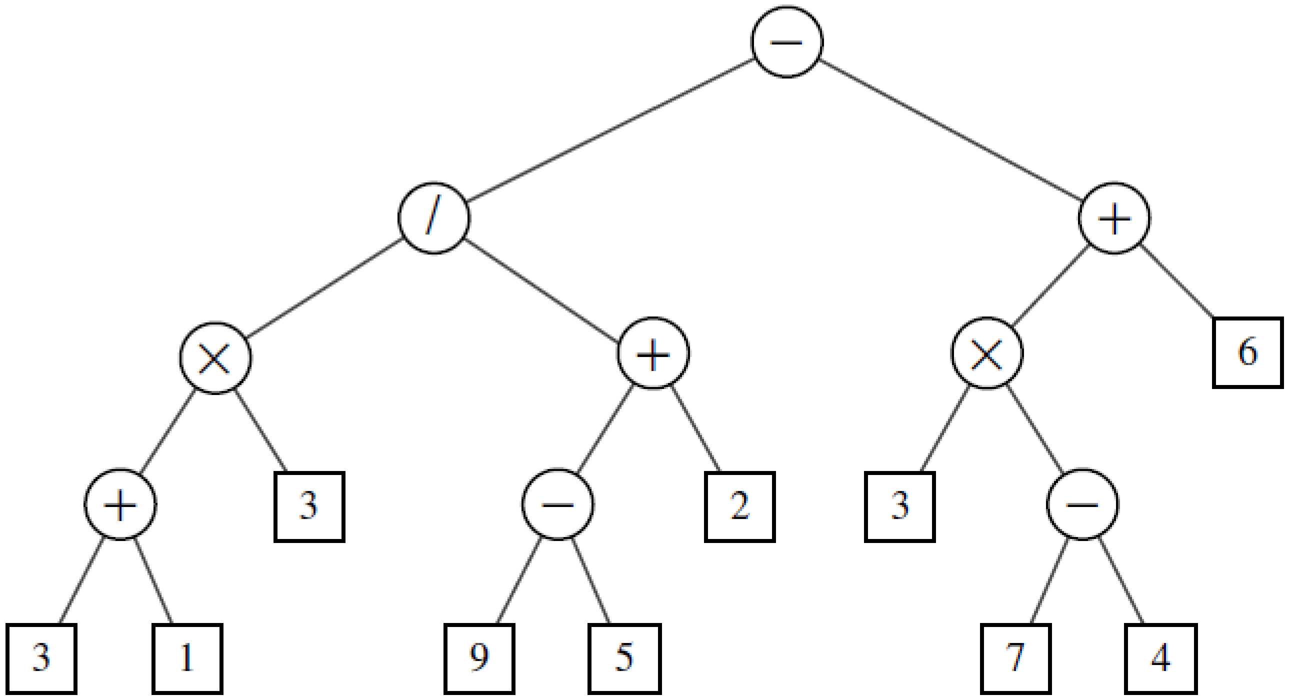
BINARY TREES

- Example: Arithmetic expression tree
- If a node is leaf, then its value is that of is variable or constant
- If a node is internal, its value is defined by applying the operator to its children
- ((3+1)x3)/((9-5)+2) – ((3x(7-4))+6)



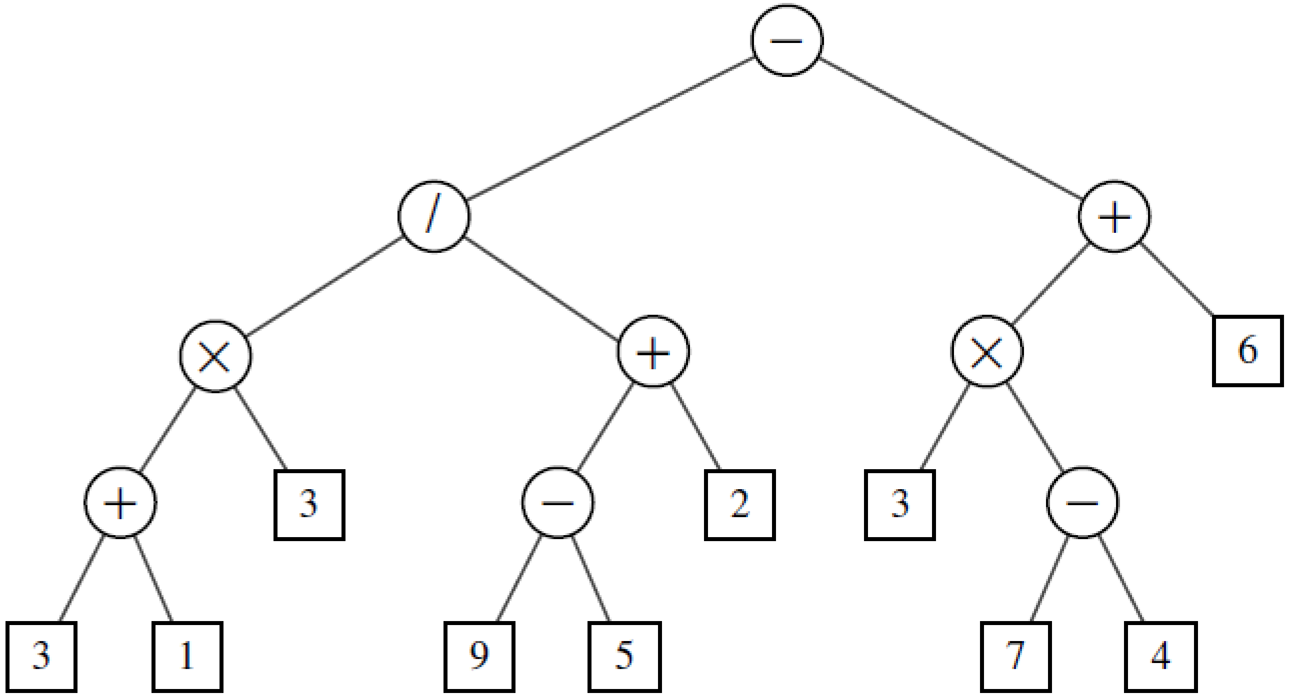
BINARY TREES

- Recursive definition
- A binary tree is either empty or consists of:
  - A node r, called the root of T, that stores an element
  - A binary tree (may be empty), called the left subtree of T
  - A binary tree (may be empty), called the right subtree of T



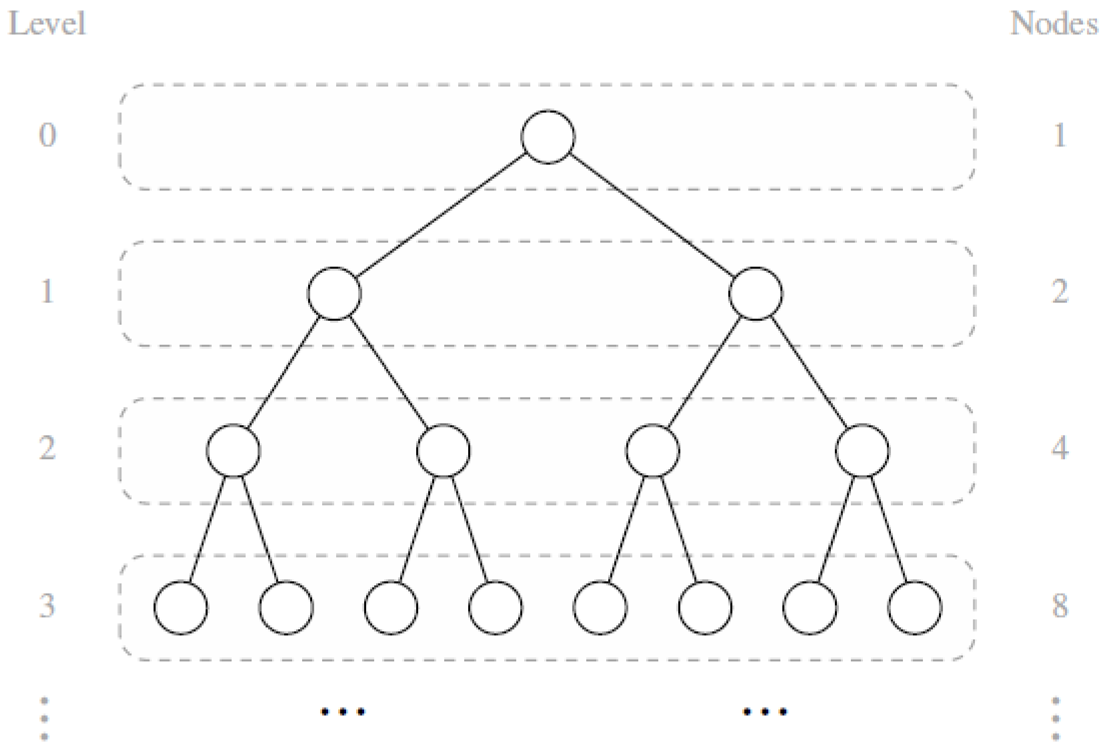
BINARY TREE ADT

- T.left(p): returns the position that represents the left child of p
- T.right(p): returns the position of the right child of p
- T.sibling(p): returns the sibling of p



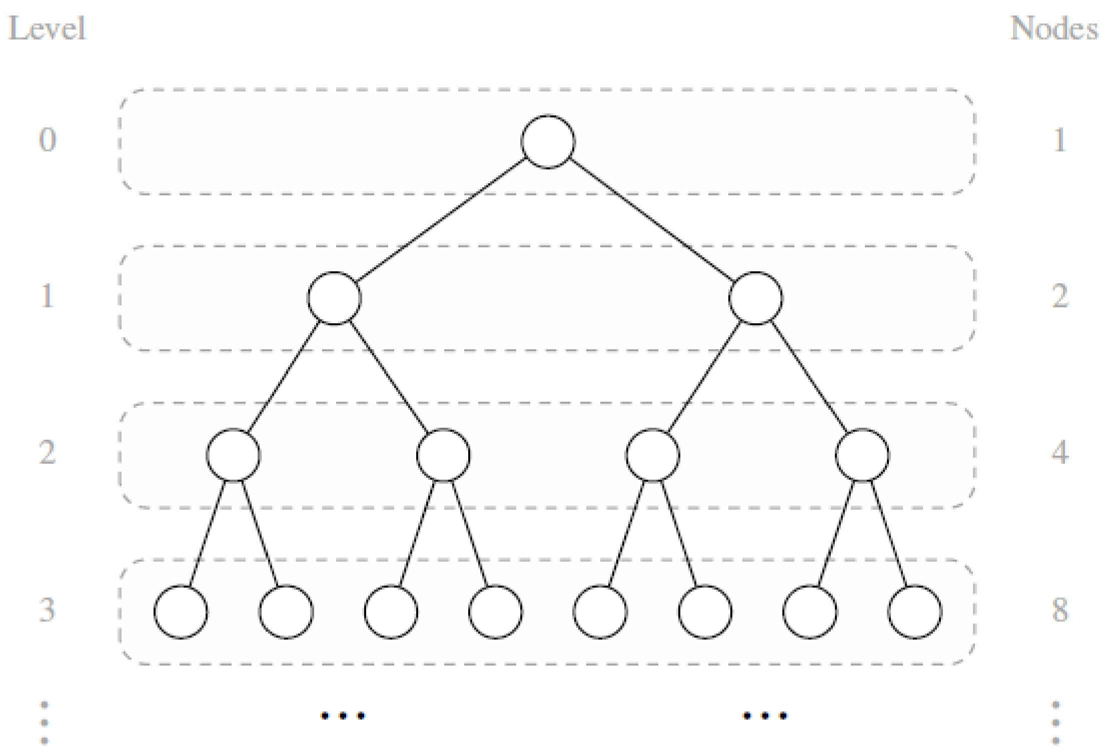
PROPERTIES OF BINARY TREES

- Depth d as level d of T
- Level 0: at most one node
- Level 1: at most 2 nodes
- Level 2: at most 4 nodes
- Level d: at most 2^d nodes



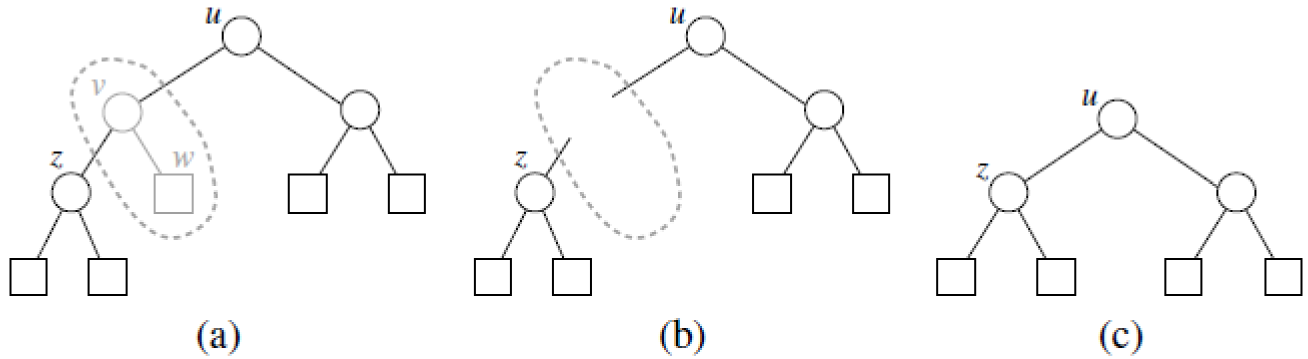
PROPERTIES OF BINARY TREES

- Let n, n_e, n_i and h denote number of nodes, number of external nodes, number of internal nodes, and height of T, then:
- $h+1 <= n <= 2^{h+1} -1$
- $1 <= n_e <= 2^h$
- $h <= n_i <= 2^h -1$
- $Log(n+1) -1 <= h <= n-1$



PROPERTIES OF BINARY TREES

- In a proper binary tree T, with n_e external nodes and n_i internal nodes, we have $n_e = n_i + 1$
- Justification:
- Case 1) if T has only one node v, remove v and place it to the external-node pile, we have $n_e = n_i + 1$
- Case 2) Otherwise, remove from T an external node w and its parent v.
  - place w on external-node pile and v on internal-node pile.
  - If v has a parent u, reconnect u with the former sibling z of w.
  - Repeat until case 1 appears



SUMMARY

- What is A tree?
  - If T is non-empty, 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
- Siblings(兄弟结点): two nodes that are children of the same parent
- External node（外部结点）: if node v has no children
- Internal node（内部结点）: if node v has one or more children
- External nodes: a.k.a. **leaves** （叶结点）
- Edge: pair of nodes (u,v) such that u is the parent of v, or vice versa.
- Path: sequence of nodes such that any two consecutive nodes in the sequence form an edge
- Depth of a tree?
  - If p is the root then depth of p is 0
  - Otherwise, the depth of p is one plus the depth of the parent of p
- Height of a tree?
  - If p is a leaf, its height is 0
  - Otherwise, the height of p is one more than the maximum of the heights of p's children
- Binary Tree?
  - A binary tree is either empty or consists of:
    - A node r, called the root of T, that stores an element
    - A binary tree (may be empty), called the left subtree of T
    - A binary tree (may be empty), called the right subtree of T

SMALL QUIZ FOR THIS WEEK:

- A stream of 1s and 0s are coming.
- At any time, we have to tell that the binary number from the 1s and 0s is divisible by 3 (or not)
- For example:
  - 1 – not divisible
  - 11 – divisible
  - 110 – divisible
  - 1100 – divisible
- Try to write an algorithm that check any random binary number

THANKS

See you in the next session!