Data Structures

Augustin Cosse.

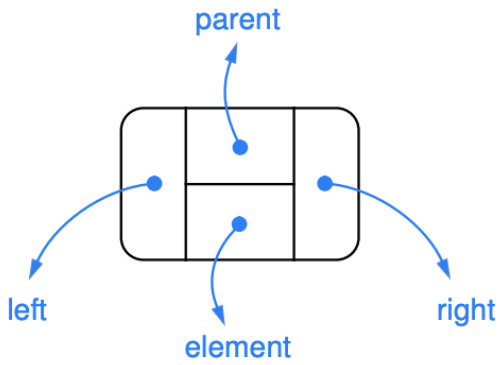
Spring 2021

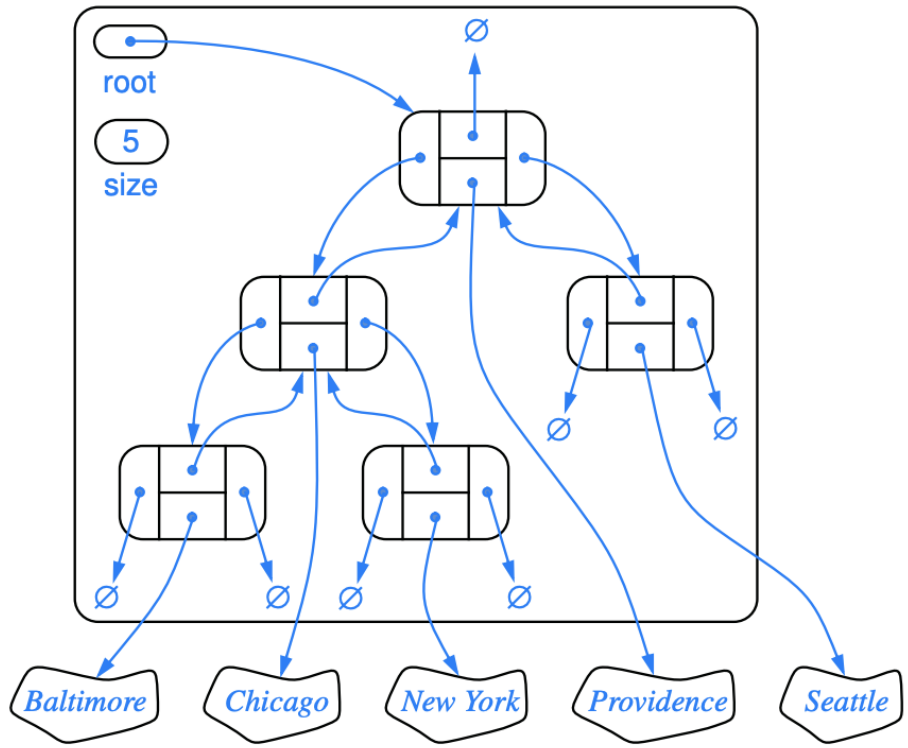
April 7, 2021

- So far, we focused on the definition of abstract classes for trees.
- Although those classes provide a great deal of support, they cannot be instantiated
- There are several choices for the internal representation of trees. We will begin with binary trees

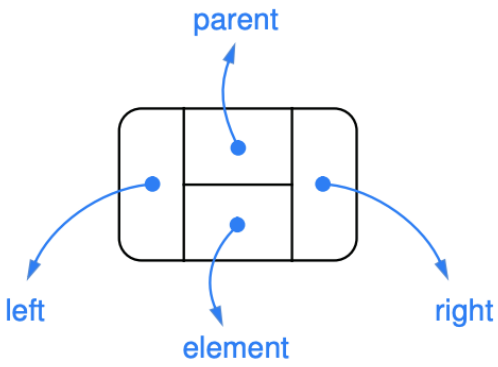
- A natural way to realize a binary tree T is to use a linked structure with a node that maintains a reference to the element stored at a position p and to the nodes associated to the children and parents of p.

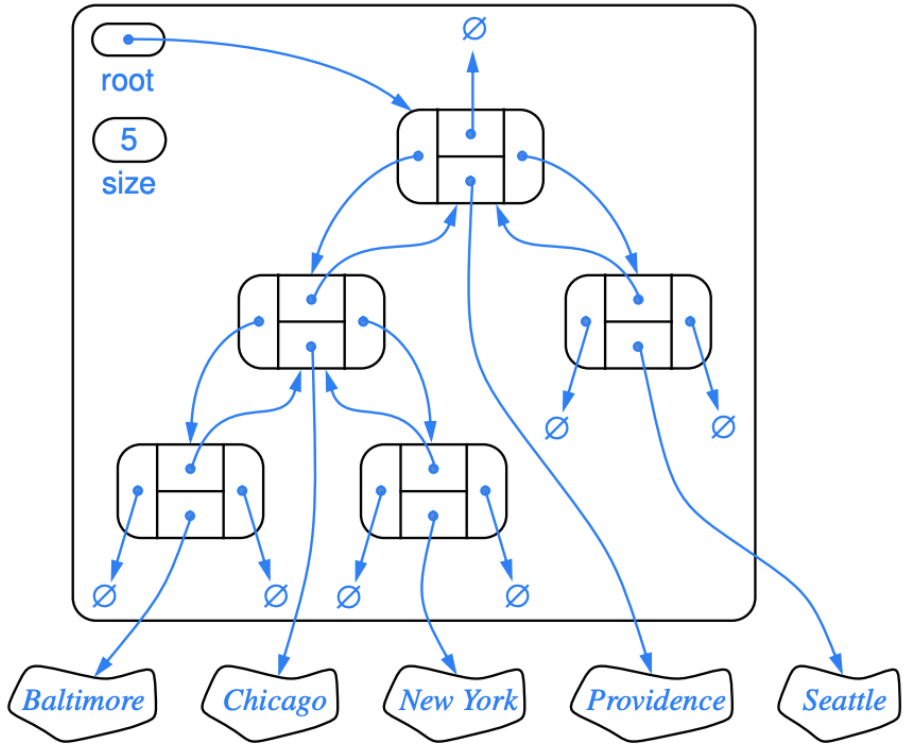
- If p is the root of T, then the parent field of p is null. Likewise, if p does not have a left child (resp. a right child), the associated field is null





- The tree itself maintains an instance variable storing a reference to the root node (if any) and a variable, called **size**, that represents the overall number of nodes of T





- The **Tree** and **BinaryTree** interfaces define a variety of methods for inspecting an existing tree, yet so far, we do not have any update method.

- Presuming that a newly created tree is empty, we would like to have means for changing the structure or content of the tree

- In the case of a linked binary tree, we will consider the following update methods

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 the previously stored element
attach(p,T_1,T_2)	Attach 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 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

- The complete set of methods on the previous two slides can all be implemented in $O(1)$ worst case time
- The most complex methods are **attach** and **remove** due to the case analyses involving the various parent-child relationships and boundary conditions
- As we did when implementing the LinkedPositionalList class, we will use a non public nested Node class to represent each of the nodes and to serve as a position of the public interface
- A node will maintain references to its parent, its left child and its right child (any of which might be null)

- The tree maintains a reference to the root node (possibly null) and a count of the number of nodes in the tree
- As for the list, we will also maintain a **validate** method that can be called any time a position is received as a parameter, to ensure that it corresponds to a valid node
- In the case of the linked tree, we adopt a convention in which we set a node's parent pointer to itself when it gets removed from the tree so that it can later be recognized as an invalid position

Linked Binary Tree (Nested Node class)

```
public class LinkedBinaryTree<E>
                    extends AbstractBinaryTree<E> {
//instance variables + nested Node class
protected static class Node<E> implements Position<E> {
  private E element;
  private Node<E> parent;
  private Node<E> left;
  private Node<E> right;
/** Nested Node Class (Part I) */
  public Node(E e, Node<E> above,
              Node<E> leftChild, Node<E> rightChild) {
    element = e;
    parent = above;
    left = leftChild;
    right = rightChild;}
// To be continued
}
```

Linked Binary Tree (Nested Node class)

```
public class LinkedBinaryTree<E>
                    extends AbstractBinaryTree<E> {
//nested Node class (part I)
protected static class Node<E> implements Position<E> {
 // Part II: accessor and update methods
  public E getElement( ) { return element; }
  public Node<E> getParent( ) { return parent; }
  public Node<E> getLeft( ) { return left; }
  public Node<E> getRight( ) { return right; }

  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; }}
```

- The nested node class implements the **Position** interface
- It also defines a method **createNode** that can be used to return a new **Node** instance
- Such a design uses what is known as factory method pattern, allowing use to later subclass our tree in order to use a specialized node pattern (i.e. we will be able to implement other types of trees with nested node classes that extends the node class of BinaryTree)

Linked Binary Tree (Part I)

```
public class LinkedBinaryTree<E>
                    extends AbstractBinaryTree<E> {
/** 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( ) { }
```

Linked Binary Tree (Part II)

```
public class LinkedBinaryTree<E>
                    extends AbstractBinaryTree<E> {
/** 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) // defunct node
      throw new IllegalArgumentException
                    ("p is no longer in the tree");
    return node;}
// accessor methods (not in AbstractBinaryTree)
  public int size( ) { /* num of nodes*/
    return size;}
  public Position<E> root( ) { /* root position*/
    return root;}
```

Linked Binary Tree (Part III)

```
/** Returns Position of parent */
public Position<E> parent(Position<E> p) throws
                    IllegalArgumentException {
  Node<E> node = validate(p);
  return node.getParent( );}
/** Returns Position of left child */
  public Position<E> left(Position<E> p) throws
                  IllegalArgumentException {
    Node<E> node = validate(p);
    return node.getLeft( );}

/** Returns Position of right child */
  public Position<E> right(Position<E> p) throws
                    IllegalArgumentException {
    Node<E> node = validate(p);
    return node.getRight( );}
```

Linked Binary Tree (Part IV)

```
/** Places e at root of empty tree*/
  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 new left child of Position p */
  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;}
```

Linked Binary Tree (Part V)

```
/** Creates right child */
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 */
public E set(Position<E> p, E e) throws
                IllegalArgumentException {
  Node<E> node = validate(p);
  E temp = node.getElement( );
  node.setElement(e);
  return temp;}
```

Linked Binary Tree (Part VI)

```
/** Attaches T1 and T2 as left and right subtrees of 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
    t1.root.setParent(node);
    node.setLeft(t1.root);
    t1.root = null;
    t1.size = 0;
}
  if (!t2.isEmpty( )) { // attach t2 as right subtree
    t2.root.setParent(node);
    node.setRight(t2.root);
    t2.root = null;
    t2.size = 0;
}}
```

Linked Binary Tree (Part VII)

- We conclude the implementation a method that removes a node and reconnect its parent with its left subtree

```
public E remove(Position<E> p) throws
                    IllegalArgumentException { // Part I
  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( ));
  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);}
```

Linked Binary Tree

- In **remove()**, we use conditional branching. I.e. the fact that Java provide a compressed syntax for the expression

```
boolean accessAllowed;
double age = 10;
if (age > 18) {accessAllowed = true;} else
{accessAllowed = false;}
```

as

```
boolean accessAllowed = (age > 18) ? true : false;
```

In short, we rely on the syntax

```
boolean result = condition ? value1 : value2;
```

Linked Binary Tree

- We conclude the implementation a method that removes a node and reconnect its parent with its left subtree

```
public E remove(Position<E> p) throws
                    IllegalArgumentException {
// Part II
  size--;
  E temp = node.getElement( );
  node.setElement(null); // help garbage collection
  node.setLeft(null);
  node.setRight(null);
  node.setParent(node); // defunct node convention
  return temp;}// end of remove method
} // end of LinkedBinaryTree class
```

Runtime

- Among the methods that we implemented above:
  - The **size** method uses an instance variable storing the number of nodes of a tree and therefore only takes $O(1)$ time. The same is true for the **isEmpty** method
  - The accessor methods root, left, right and **parent** which are implemented in **LinkedBinaryTree** take $O(1)$ each. The **sibling, children** and **numChildren** which are defined in **AbstractBinaryTree** also rely on a constant number of calls to those accessor methods and hence run in $O(1)$ as well.
  - The **isInternal** and **isExternal** methods inherited from the **AbstractTree** class rely on a call to **numChildren** and hence run in $O(1)$ time. The **isRoot** method, also implemented in **AbstractTree** relies on a comparison to the result of the **root** method and hence also runs in $O(1)$

Runtime

- Among the methods that we implemented above:
  - The update method **set** runs in $O(1)$ time and so run all the methods **addRoot, addLeft, addRight, attach** and **remove**
  - We already analyzed the methods **depth** and **height** when discussing the AbstractTree class and we concluded that **depth** runs in $O(d_p + 1)$ for a position p at depth d_p and that **height** runs in $O(n)$ for the root

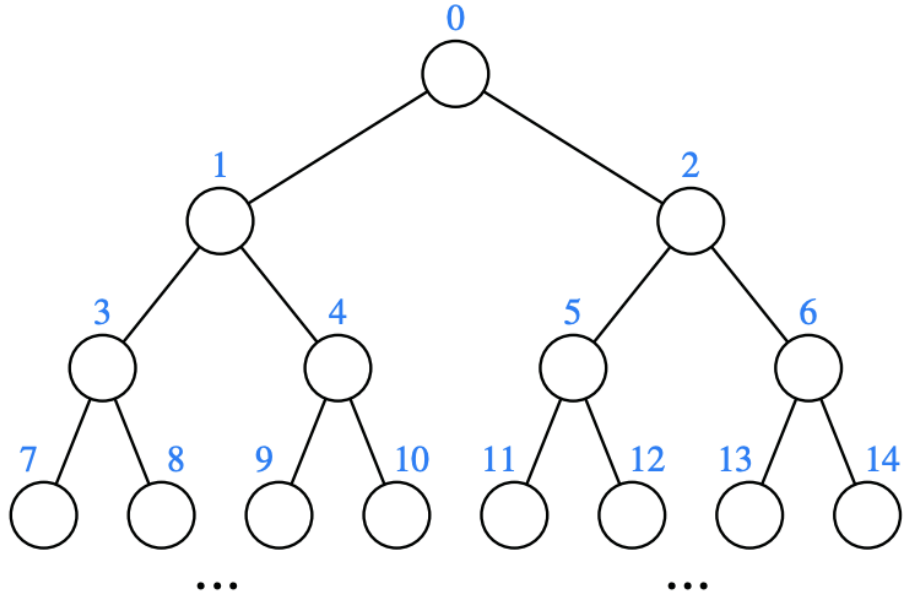
Method	Running Time
size, isEmpty	$O(1)$
root, parent, left, right, sibling, children, numChildren	$O(1)$
isInternal, isExternal, isRoot	$O(1)$
addRoot, addLeft, addRight, set, attach, remove	$O(1)$
depth(p)	$O(d_p+1)$
height	$O(n)$

Array based representation

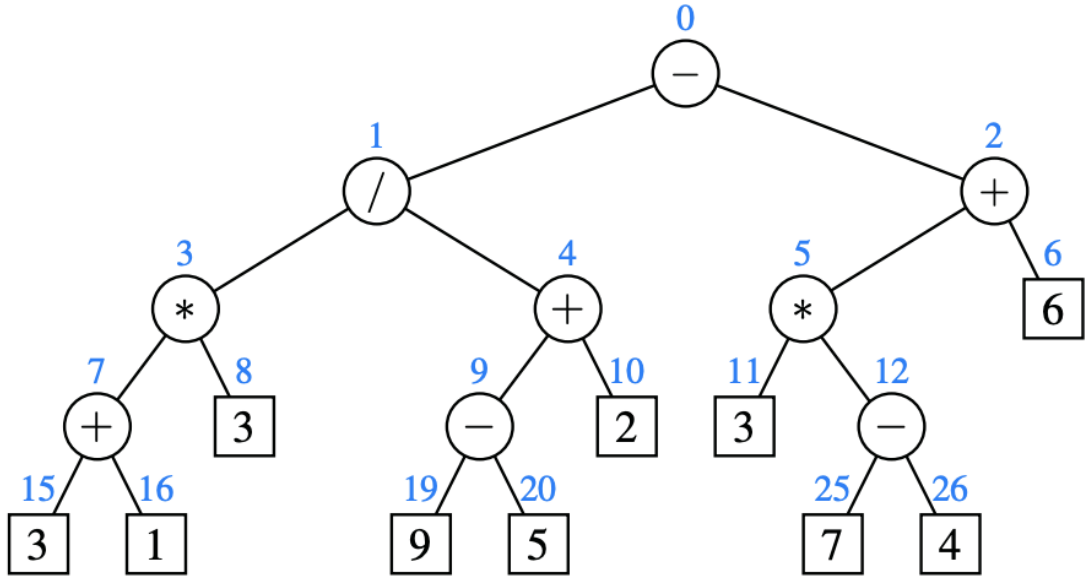
- Just as for Positional Lists, we can alternatively represent our binary tree by relying on arrays
- For every position p of T, let $f(p)$ be the integer defined as follows
  - If p is the root of T, then $f(p) = 0$
  - If p is the left child of position q, then $f(p) = 2 * f(p) + 1$
  - If p is the right child of position q, then $f(p) = 2 * f(p) + 2$
- The numbering function f is known as the level numbering of the positions in a binary tree for it numbers the positions on each level of the tree in increasing order from left to right
- Note that the numbering is based on the potential positions within the tree and not the actual shape.

Array based representation

(a)

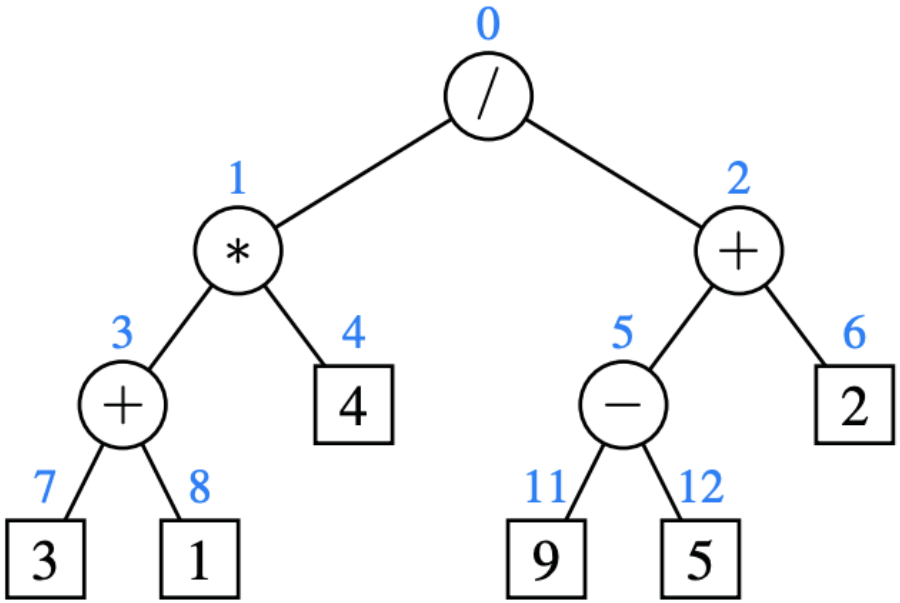


(b)



Array based representation

- The level numbering function f suggests a representation of a binary tree by means of an array based structure A, with the elements at position p of T stored at index $f(p)$ of the array



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

Array based representation

- An advantage of the array based representation is that a position p can be encoded by the single integer $f(p)$ and consequently, that position based methods such as **root**, **parent**, **left** and **right** can be implemented using simple arithmetic operations on the number $f(p)$
- Based on the level numbering formula, the left child of a node p has index $2f(p) + 1$, the right child has index $2f(p) + 2$ and the parent has index $\lfloor (f(p) - 1)/2 \rfloor$
- The space usage of an array based representation depends greatly on the shape of the tree
- If we let n to denote the number of nodes and let f_M denote the maximum value of $f(p)$ over all nodes p of T. The array requires size $1 + f_M$ as we label the root starting from $f(r) = 0$

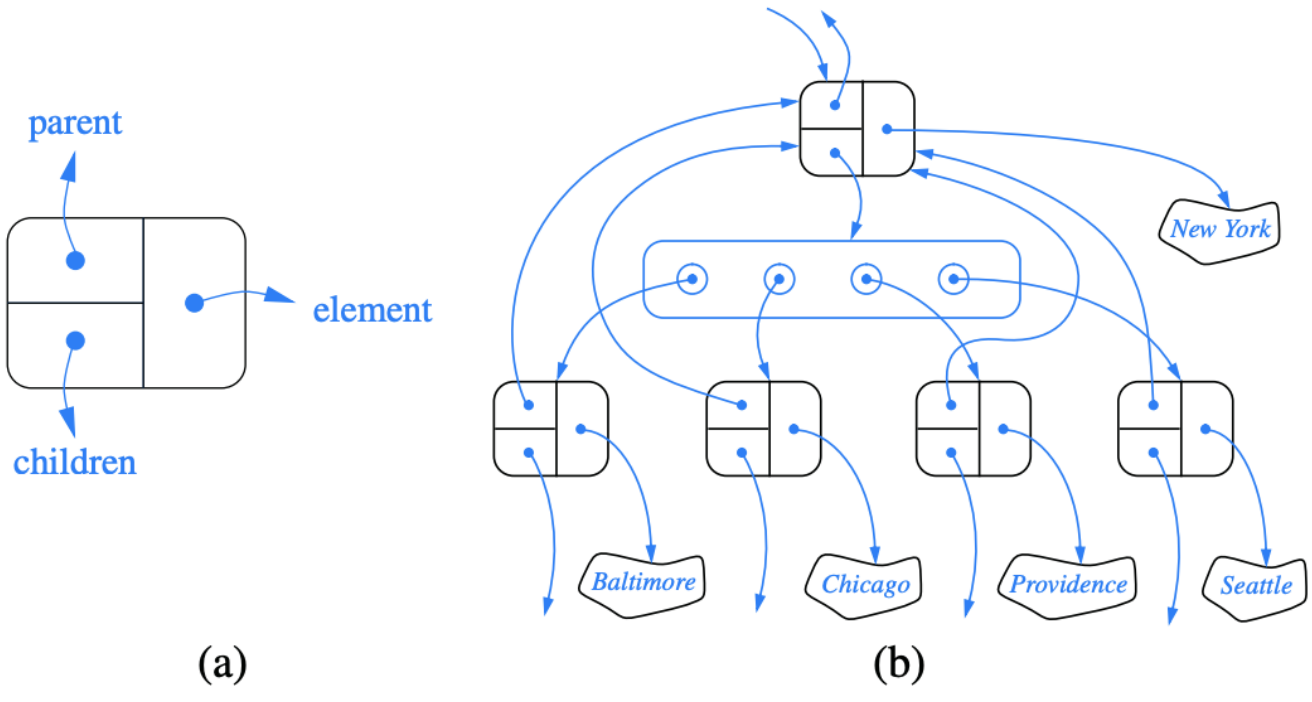
Array based representation

- Note that the array may have a number of empty cells. In fact, in the worst case, one can check that $N = 2^n - 1$ (why?)
- We will also study a special class of binary trees called heaps for which $N = n$
- In spite of the worst case memory ussage which is clearly suboptimal, there are still applications for which the the array representation of binary trees is efficient (e.g. heaps). For binary trees though, the exponential worst case space requirement is prohibitive
- Another drawback of the array representation is that many array operations for trees cannot be efficiently supported. For example, removing a node and promoting its child takes $O(n)$ runtime because it is not just the child that moves locations within the array but all the descendants of that child

From binary trees to general trees

- For a general tree, there is a priori no limit on the number of children that a node may have

- A natural way to realize a general tree as a linked structure is to have each node store a single container of references to its children. For example, a children field of a node can be an array or list of references to the children of the node (if any)



From binary trees to general trees

- The performance of the implementation of a general tree using a linked structure is given below
- We let c_p denote the number of children of the node at position p and d_p its depth

Method	Running Time
size, isEmpty	$O(1)$
root, parent, isRoot, isInternal, isExternal	$O(1)$
numChildren(p)	$O(1)$
children(p)	$O(c_p+1)$
depth(p)	$O(d_p+1)$
height	$O(n)$

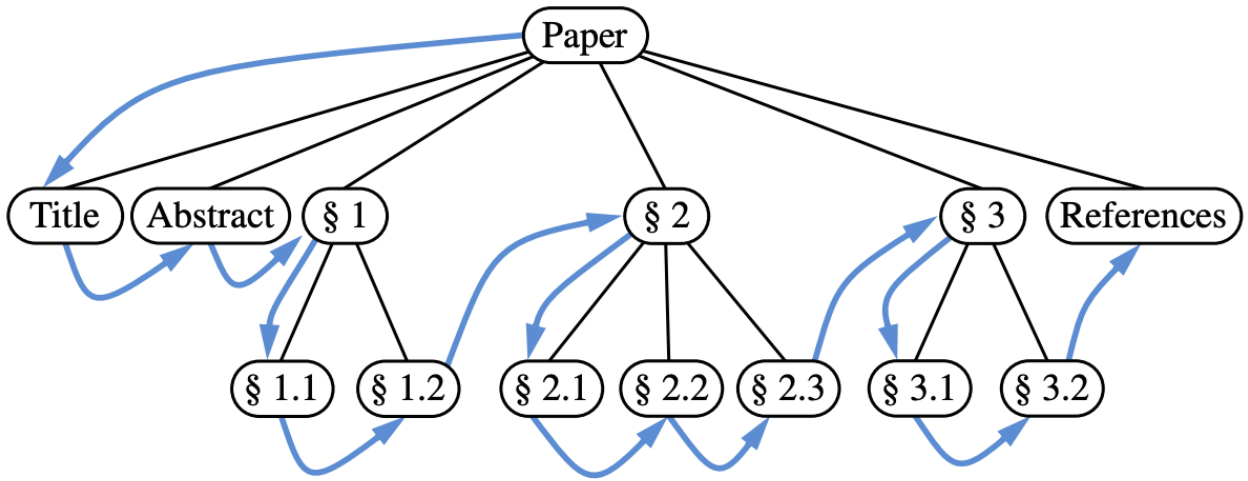
Tree Traversal Algorithms

- A traversal of a tree T is a systematic way of accessing or "visiting" all the positions of T.
- The specific action associated with the "visit" of a position depends on the application of this traversal and could involve anything from incrementing a counter to performing some complex computation for p

Tree Traversal Algorithms

- In a preorder traversal of a tree T, the root of T is visited first and then the subtrees are traversed according to the order of the children

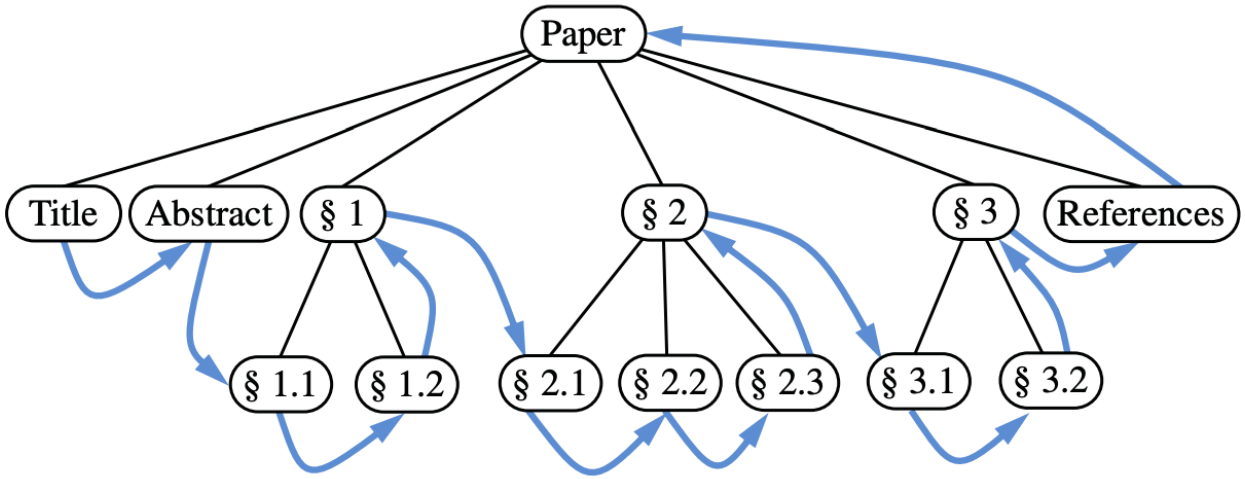
```
perform the "visit" action for position p
// happens before recursion
for each child node c in children(p) do
  preorder(c)
// recursively traverse subtree rooted at c
```



Tree Traversal Algorithms

- In a postorder traversal of a tree T, the root of T is visited first and then the subtrees are traversed according to the order of the children

```
for each child node c in children(p) do
  postorder(c)
// recursively traverse the tree rooted at c
perform the "visit" action for position p
// happens after the recursion
```



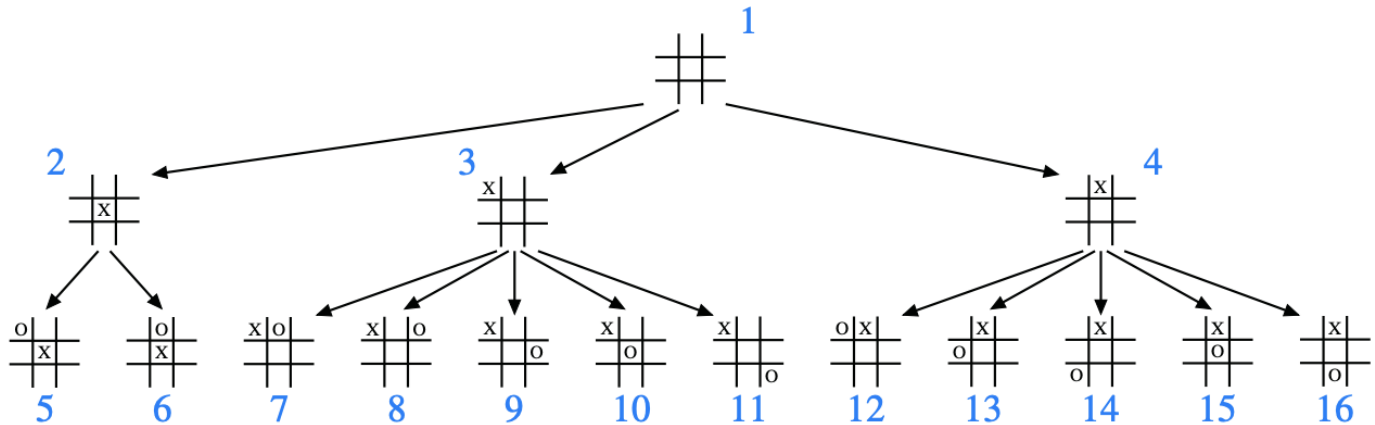
Runtime Analysis

- Both preorder and postorder are efficient ways to access all the positions of a tree
- The analysis of either of these traversal algorithms is similar to that of the algorithm height
- At each position p, the non recursive part of the traversal algorithm requires time $O(c_p + 1)$ where c_p is the number of children of p
- The overall running time for the traversal of the tree is $O(n)$ where n is the total number of positions in the tree

Breadth First Tree Traversal

- Although the preorder and postorder are two common ways of visiting the postions of a tree, another approach is to traverse a tree so that we visit all the positions at depth d before we visit the positions at depth $d + 1$. Such an approach is known as breadth-first traversal

- A breadth first traversal is a common approach used in software for playing games. A game tree represents the possible choices of moves that might be made by a player (or computer) during a game with the root of the tree being the initial configuration of the game



Breadth First Tree Traversal

- A breadth first traversal of a game tree is often performed because a computer may be unable to explore a complete game tree in a limited amount of time
- The computer will then consider all moves, then responses to those moves, going as deep as computational time allows

```
Initialize queue Q to contain root()
while Q not empty do
  p = Q.dequeue() // p is oldest entry in queue
  perform the "visit" action for position $p$
  for each child $c$ in children(p) do
    Q.enqueue(c)
// add p's children to the end of the queue
// for later visits
```

Breadth First Tree Traversal

- Note that the breadth first process is not recursive as we are not traversing entire subtrees at once
- We use a queue to produce a FIFO semantics for the order in which we visit the nodes
- The overall running time is $O(n)$ due to the n calls to enqueue and the n calls to deuque.

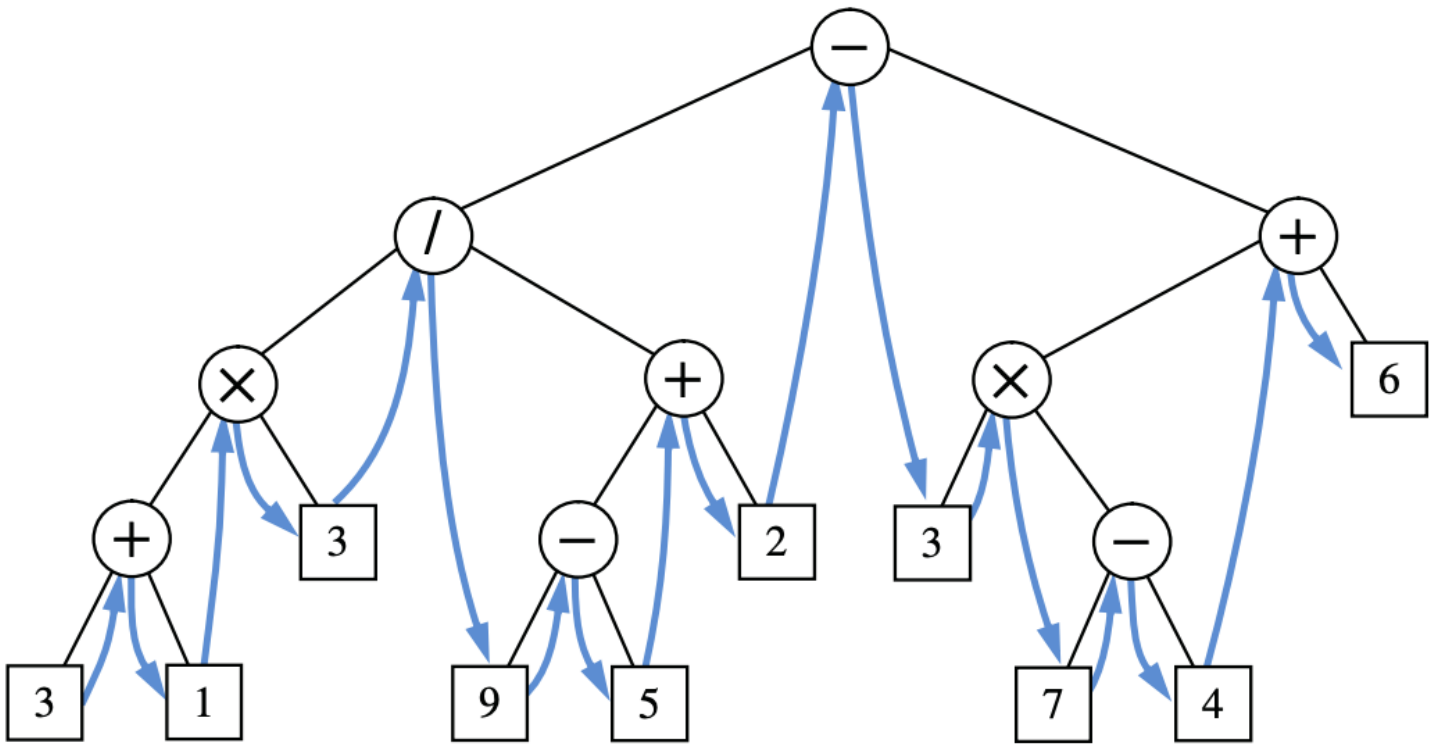
```
Initialize queue Q to contain root()
while Q not empty do
  p = Q.dequeue() // p is oldest entry in queue
  perform the "visit" action for position $p$
  for each child $c$ in children(p) do
    Q.enqueue(c)
// add p's children to the end of the queue
// for later visits
```

In Order traversal of Binary Tree

- During an in-order traversal (on binary trees), we visit a position between the traversal of its left and right subtrees
- The in-order traversal of a binary tree T can be informally viewed as visiting the nodes of T "from left to right"
- Indeed, for every position p, the inorder traversal visits p after all the positions in the left subtree of p and before all the positions in the right subtree of p

```
if p has a left child lc then
  innorder(lc)
perform the "visit" action for position p
if p has a right child rc then
  inorder(rc)
```

In Order traversal of Binary Tree



Binary Search Tree

- An important application of the inorder traversal algorithm arises when we store an ordered sequence of elements in a binary tree, defining a structure we call a binary search tree
- Let S be a set whose unique elements have an order relation. For example, S could be a set of integers. A binary search tree for S is a proper binary tree T such that, for each internal position p of T:
  - Position p stores an element of S, denoted as $e(p)$
  - Elements stored in the left subtree of p (if any) are less than $e(p)$
  - Element stored in the right subtree of p (if any) are greater than $e(p)$

Binary Search Tree