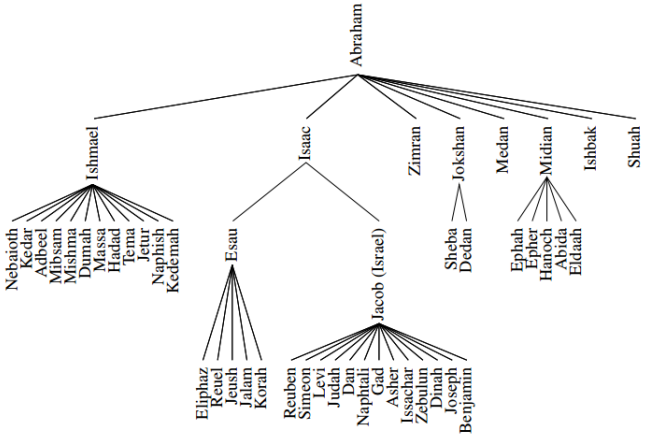
Data Structures and Algorithms
(ECEG 4171)

Chapter Five
Trees

Introdution and Terminologies

- A tree is a nonlinear data structure that stores elements hierarchically.
- Trees are ubiquitous structures in operating systems, graphical user interfaces, databases, websites, and many other computer systems.



Terminologies

- A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies 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.
- Two nodes that are children of the same parent are siblings.
- A node v is external if v has no children. External nodes are also known as leaves.
- 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. Also, a node v is a descendant of a node u if u is an ancestor of v. For example from the figure in slide 6, cs252/ is an ancestor of papers/, and pr3 is a descendant of cs016/.

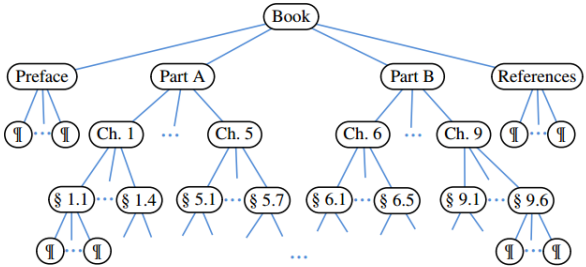
Terminologies

- The subtree of T rooted at a node v is the tree consisting of all the descendants of v in T (including v itself). In Figure given in the next slide, the subtree rooted at cs016/ consists of the nodes cs016/, grades, homeworks/, programs/, hw1, hw2, hw3, pr1, pr2, and pr3.
- 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. For example, the tree in Figure from slide 6 contains the path (cs252/, projects/, demos/, market).
- Hence, a tree can be viewed as a set of nodes and edges that connect them and there is exactly one path between two nodes.

Terminologies

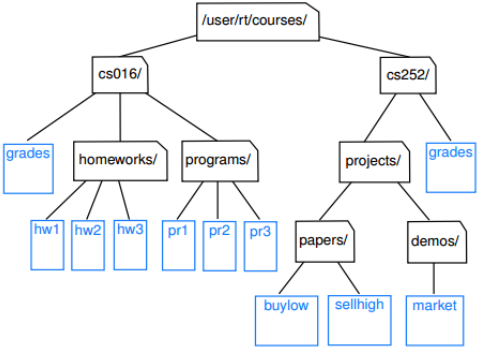
- A tree is ordered if there is a meaningful linear order among the children of each node; that is, we purposefully identify the children of a node as being the first, second, third, and so on from left to right,.

Example of ordered tree



- The tree above is an example of an ordered tree, because there is a well-defined order among the children of each node.

Tree Example



- The figure above show the hierarchical relationship between files and directories in a computer's file system represented as a tree.
- We see that the internal nodes of the tree are associated with directories and the leaves are associated with regular files.

Iterators and Iterables

- Oftentimes you want to cycle the elements of a collection, for example, to display each element.
- One way to do this is to employ an *iterator*, which is an object that implements either the `Iterator` or the `ListIterator` interface.
- Java defines the `java.util.Iterator` interface with the following two methods:

`hasNext()`:	Returns true if there is at least one additional element in the sequence , and false otherwise.
`next()`:	Returns the next element in the sequence or a `NoSuchElementException` exception is thrown if no further elements are available.
`remove()`:	Removes from the collection the element returned by the most recent call to `next()`. Throws an `IllegalStateException` if `next` has not yet been called, or if remove was already called sincethe most recent call to next.

Iterators and Iterables

- The combination of `hasNext()` and `next()` methods allows a general loop construct for processing elements of the iterator.
- For example, if `iter` denotes an instance of the `Iterator<String>` type

```
while (iter.hasNext()) {
    String value = iter.next();
    System.out.println(value);
}
```

- The `remove()` method can be used to filter a collection of elements, for example to discard all negative numbers from a data set.
- The `remove` method is not implemented for most data strucutures.

The Iterable Interface

- The `Iterator` instance supports only one pass through a collection.
- There is no way to reset the iterator back to the beginning of the sequence.
- Java defines another parameterized interface, named `Iterable`, that includes the following single method:

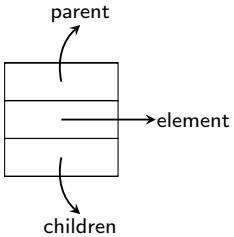
  `iterator()`: Returns an iterator of the elements in the collection.

- Each call to `iterator()` returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.
- Java's `Iterable` class also plays a fundamental role in support of the for-each loop syntax. The loop syntax,

```
for (ElementType variable : collection ) {
   loopBody // may refer to " variable"
}
```

Implementation of General Trees

- The challenging part in tree implementation is there is no a priori limit on the number of children a node may have.
- One way to realize a general rooted tree is to have each node store three references:
  - One a reference to a generic type to refer the element of a node.
  - One a reference to the parent of a node.
  - One a reference of linked list or arraylist type to refer to the children a node.




```
import java.util.List;
public class Tree<E>{
    static class Node<E>{
        E element;
        Node<E> parent;
        List<Node<E>> children;
    }
    Node<E> root;
}
```

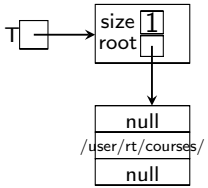
Implementation of General Trees

```
import java.util.List;
import java.util.LinkedList;
public class Tree<E>{
    public static class Node<E>{
        private E element;
        private Node<E> parent;
        private List<Node<E>> children;
        public Node(E e, Node<E> p, List<Node<E>> c){
            element = e;
            parent = p;
            children = c;
        }
    }
    private Node<E> root;
    private int size;
    public int size(){ return size; }
    public Node<E> getRoot(){ return root; }
    public void addRoot(E e){
        root = new Node<>();
        size = 1;
    }
    public boolean isRoot(Node<E> n){ return root == n; }
    public boolean isEmpty(){ return size == 0; }
    public int numChildren(Node<E> n){ return n.children.size(); }
    public boolean isExternal(Node<E> n){ return n.children == null; }
    public boolean isInternal(Node<E> n){ return n.children != null; }
    public void addChildren(Node<E> n, List<Node<E>> c)
                throws IllegalStateException {
        if(n.getChildren() != null)
            throw new IllegalStateException("n already has children.");
        n.children = c;
        size += c.size();
    }
}
```

Implemenatation of General Trees

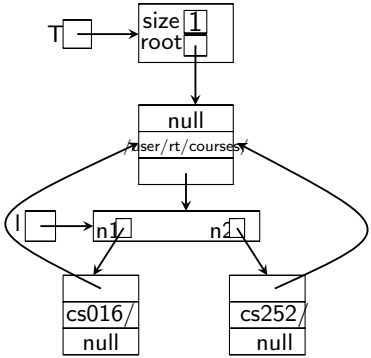
Now let's create the tree given in Figure above for two levels:

```
Tree<String> T = new
Tree("/user/rt/courses/");
```



Implementation of General Trees

```
Node<String> n1 =
new Node<>("cs016/", T.getRoot(), null);

Node<String> n2 =
new Node<>("cs252/", T.getRoot(), null);

List<Node<String>> l = new LinkedList<>();
l.add(n1);
l.add(n2);
T.addChildren(l);
```



Computing Depth and Height

- The **depth** of a node n is the number of ancestors of n, other than n itself. This definition implies that the depth of the root of T is 0.
- Recursive definition:
  - If n is the root, then the depth of n is 0.
  - Otherwise, the depth of n is one plus the depth of the parent of n.

```
public int depth(Node<E> n) {
    if (isRoot(n)) return 0;
    else return 1 + depth(n.getParent());
}
```

- The **height** of a tree is equal to the maximum of the depths of its nodes (or zero, if the tree is empty).
- Also, the height of a node n is the maximum path from n to its children.

Computing Depth and Height

- Recursive algorithm to compute the height of a subtree rooted at a given node n

```
public int height(Node<E> n) {
    int h = 0; // base case if n is external
    for (Node<E> c : n.children())
        h = Math.max(h, 1 + height(c));
    return h;
}
```

- If the method is initially called on the root of T, it will be called once for each node of T.
- This is because the root eventually invokes the recursion on each of its children, which in turn invokes the recursion on each of their children, and so on.
- Running Time: Let T be a tree with n nodes, and let c_{n_i} denote the number of children of a node n_i of T. Then, overall running time is,

$$O(\sum_{n_i}(c_{n_i}+1)) = O(n+\sum_{n_i} c_{n_i}) = O(n)$$

Tree Traversal Algorithms

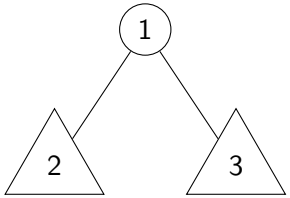
- A traversal of a tree T is a systematic way of accessing, or visiting, all the nodes of a tree.
- Because, all nodes are connected via edges (links) we always start from the root (head) node.
- There are three ways which we use to traverse a tree:
  - Preorder traversal
  - Postorder traversal
  - Breadth-first traversal

Preorder Traversal

- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Alogirthm:

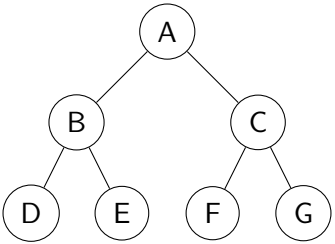
1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.



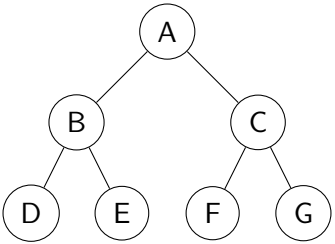
- Recursive implementation

```
public void preorder(Node<E> n){
    visit (n);
    if ( isInternal (n)){
        for(Node<E> c : n.children())
            preorder(c);
    }
}
```

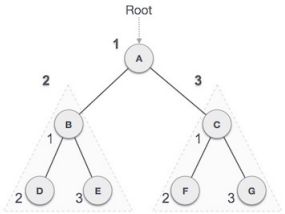
Preorder Traversal Example



Preorder Traversal Example



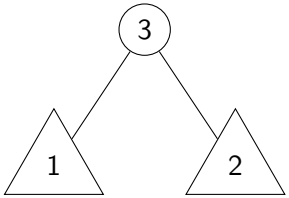
The preoder traversal of the tree is given as:



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed in pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$.

Postorder traversal

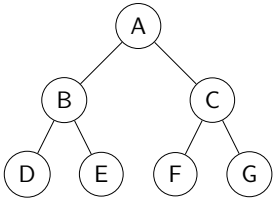
- In this traversal method, first we traverse the left subtree, then the right subtree and finally the root node.
  Alogirthm:
  1. Until all nodes are traversed:
  2. Recursively traverse left subtree
  3. Recursively traverse right subtree
  4. Visit root node



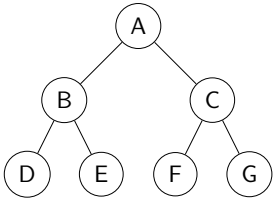
Recursive implementation:

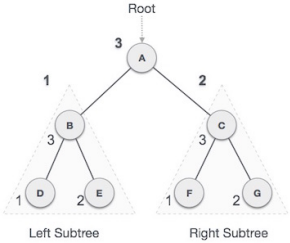
```
public void postorder(Node<E> n){
    if( isInternal (n)){
        for(Node<E> c : n.children())
            postorder(c);
    }
     visit (n);
}
```

Postorder Traversal Example



Postorder Traversal Example

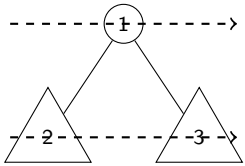




We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be:
$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Breadth-first (Level Order) Traversal

- Visits the root, then each element on the first level, then each element on the second level, and so forth:

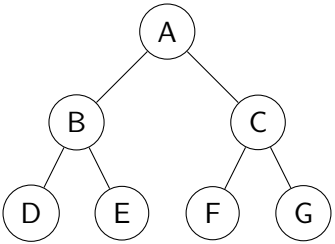


Alogirthm:

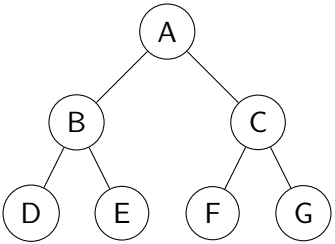
1. Initialize a queue
2. Enqueue the root
3. Repeat steps 4–6 until queue is empty
4. Dequeue node x from the queue
5. Visit x
6. Enqueue all the children of x in order

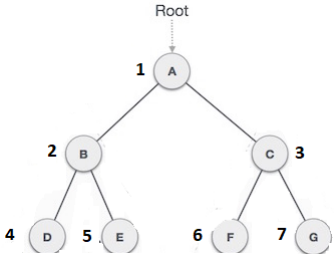
```
public void levelorder (Node<E> n){
    Queue<Node<E>> Q = new LinkedList<>();
    Q.add(n);
    while (!Q.isEmpty()){
        Node<E> t = Q.remove();
        visit (t);
        if ( isInternal (t)){
            for (Node<E> c : t.children())
                Q.add(c);
        }
    }
}
```

Beardth-first Traversal Example



Beardth-first Traversal Example





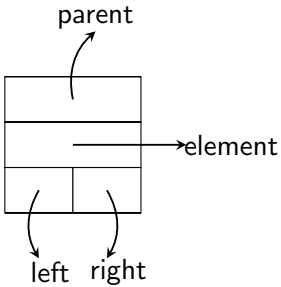
We start from *level 0* and visit all nodes in that level from *left* to *right*. Then we follow the same procedure for levels 1 and 2. Hence, the traversal order is:
$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$

Binary Trees

- A binary tree is an ordered tree with the following properties:
  1. Every node has at most two children.
  2. Each child is labeled as being either a left child or right child.
  3. A left child precedes a right child in the order of children of a node.
- A binary tree is said to **proper** if every internal node has exactly two children.

Binary Tree Implemntation

- A natural way to realize a binary tree T is to use a *linked structure*




```
import java.util.List;
public class Tree<E>{
    static class Node<E>{
        E element;
        Node<E> parent;
        Node<E> left, right;
    }
    Node<E> root;
}
```

- Just like we did for general trees, we can encapuslate instance variables and add getters and setters.

Binary Tree Implementation

```
import java.util.*;
public class BinaryTree<E>{
    public static class Node<E>{
        private Node<E> parent;
        private E element;
        private Node<E> left;
        private Node<E> right;

        public Node(Node<E> p, E e, Node<E> l, Node<E> r){
            parent = p;
            element = e;
            left = l;
            right = r;
        }
        public Node<E> getParent(){ return parent; }
        public E getElement(){ return element; }
        public Node<E> getLeft(){ return left; }
        public Node<E> getRight(){ return right; }

        public void setParent(Node<E> p){ parent = p; }
        public void setElement(E e){ element = e; }
        public void setLeft(Node<E> l){ left = l; }
        public void setRight(Node<E> r){ right = r; }
    }
    private Node<E> root;
    private int size = 0;
    public int size(){ return size; }
    public boolean isEmpty(){ return size == 0; }
    public Node<E> root(){ return root; }
    public BinaryTree(E e){
        root = new Node(null, e, null, null);
        size = 1;
    }
    /** Returns the number of children of Node n. */
    public int numChildren(Node<E> n){
        int count = 0;
        if(n.getLeft() != null)
            count++;
        if(n.getRight() != null)
            count++;
        return count;
    }
    public boolean isInternal(Node<E> n){
        return (numChildren(n) > 0);
    }
    /** Returns an iterable collection of the nodes representing
    n's children. */
    public Iterable<Node<E>> children(Node<E> n){
        List<Node<E>> c = new LinkedList<>();
        if(n.getLeft() != null) c.add(n.getLeft());
        if(n.getRight() != null) c.add(n.getRight());

        return c;
    }
}
```

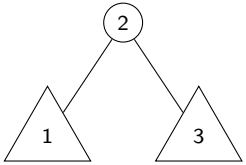
```
    public Node<E> addLeft(Node<E> n, E e) throws IllegalStateException{
        if (n.getLeft() != null)
         throw new IllegalStateException ("n has already a left child.");
        Node<E> child = new Node(n, e, null, null);
        n.setLeft(child);
        size++;
        return child;
    }
    public Node<E> addRight(Node<E> n, E e) throws IllegalStateException{
        if (n.getRight() != null)
         throw new IllegalStateException ("n has already a right child.");
        Node<E> child = new Node(n, e, null, null);
        n.setRight(child);
        size++;
        return child;
    }
}
```

Inorder traversal of binary trees

- The inorder traversal algorithm specifically applies for a binary trees
- The inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T from left to right.

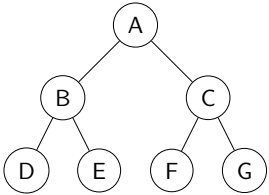
Alogirthm:

1. Recursively traverse the left subtree
2. Visit the root
3. Recursively traverse the left subtree

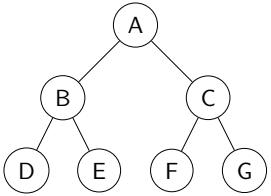


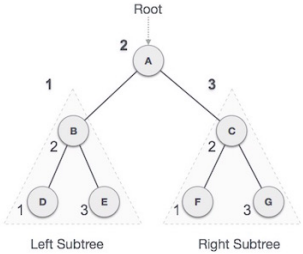

```
public void inorder(Node<E> n){
    if(n == null) return;
    inorder(n.getLeft());
     visit (n);
    inorder(n.getRight());
}
```

Inorder Traversal Example



Inorder Traversal Example

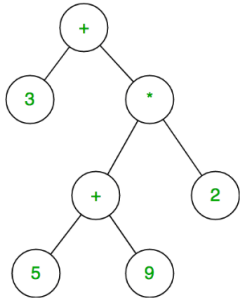




We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be:
$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$.

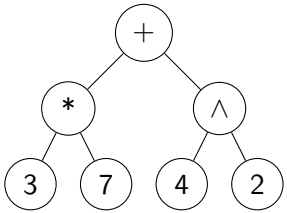
Expression Trees

- Expression tree is binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.
- For example the expression tree for $3 + ((5 + 9) * 2)$



Expression Trees

- The inorder traversal of expression trees produces **infix expression**, the preorder traversal produces **prefix expression**, the postorder traversal produces **postfix expression**
- For example, for the expression $3 * 7 + 4 \wedge 2$



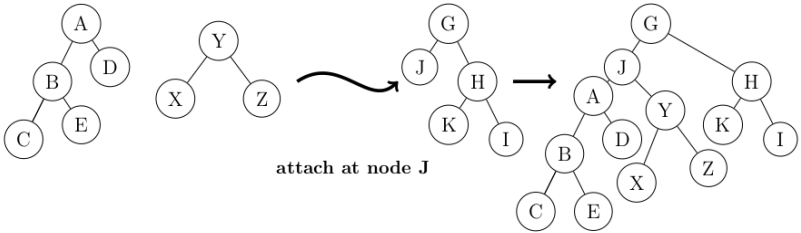
Preorder: $+ * 3\,7 \wedge 4\,2$ (easier for computers)
Inorder: $3 * 7 + 4 \wedge 2$ (easier for humans)
Postorder: $3\,7 * 4\,2 \wedge +$ (easier for computers)

When to Use Pre-, Post-, In- Order Traversals

- Using the right traversal algorithm brings a considerable speed difference.
- If you know you need to explore the roots before inspecting any leaves, you pick pre-order because you will encounter all the roots before all of the leaves.
  - E.g. Preoder is used to make a duplicate of a tree and also to produce prefix expressions.
- If you know you need to explore all the leaves before any nodes, you select post-order because you don't waste any time inspecting roots in search for leaves.
  - E.g. Postorder traversal is a natural way to sum total disk space and also for freeing deleting an entire tree.
- If you know that the tree has an inherent sequence in the nodes, and you want to flatten the tree back into its original sequence, then an in-order traversal should be used.
  - E.g. in Binary search trees it is used to generate a sequence of elements in sorted order.

Exercise

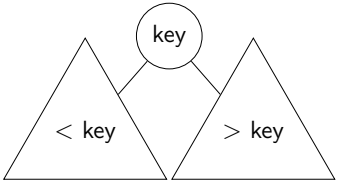
Write a method with signature `public void attach(Node<E> n, BinaryTree<E> t1, BinaryTree<E> t2)` that attaches two binary trees *t1* and *t2* as right and left subtrees of a binary tree into its external node *n*. (Note: make sure *n* is a leaf.)



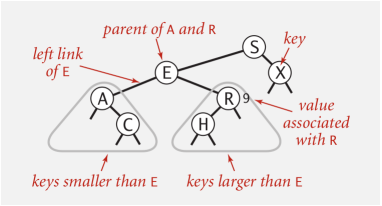
```
public void attach(Node<E> n, BinaryTree<E> t1, BinaryTree<E> t2) throws IllegalArgumentException{
    if( isInternal (n)) throw new IllegalArgumentException("n must be a leaf");
    size = size + t1.size() + t2.size();
    if (!t1.isEmpty()){
        t1.root.setParent(n);
        n.setLeft(t1.root);
        t1.root = null;
        t1.size = 0;
    }
    if (!t2.isEmpty()){
        t2.root.setParent(n);
        n.setRight(t2.root);
        t2.root = null;
        t2.size = 0;
    }

}
```

Binary Search Trees

- Definition. A BST is a binary tree in symmetric order, that is, each node has a key, and every node's key is:
  - Larger than all keys in its left subtree.
  - Smaller than all keys in its right subtree.



- E.g. A BST with character keys.



- A BST is an example of a Dictionary (*a.k.a* Symbole table).

The Dictionary ADT

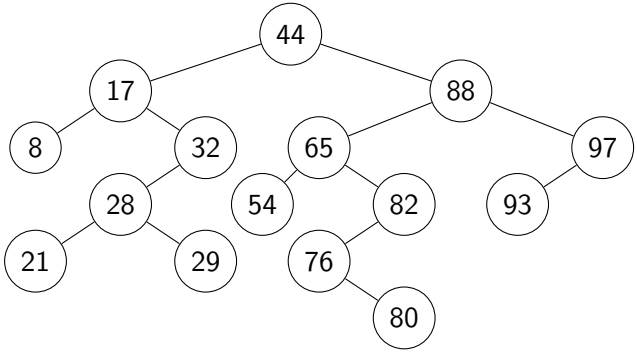
- A dictionary is an ADT whose elements are key-value pairs and stores elements indexed using keys. All dictionaries support the insert, search, and delete operations.
- Applications of dictionaries include:

application	*purpose of search*	*key*	*value*
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
web search	find relevant web pages	keyword	list of page names
compiler	find type and value	variable name	type and value

- BSTs implement efficiently the following dictionary operations:

`get(k)`	find the key `k` in the data structure, or determine that `k` is not there.
`put(k, v)`:	insert the key k and the value v associated with it into the data structure. If it's already there, replace its value with v.
`delete(k)`:	delete the node with key k from the data structure if it's there.

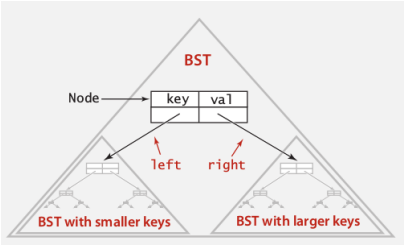
- An example of BST with Integer keys and values omitted for simplicity.



BST Implementation

- In Java, we can implement a BST as a reference to a root node.
- A `Node` is composed of five fields:
  - A `Key` and a `Value`.
  - A reference to the left and right subtrees
  - A node count

```
private class Node{
   private K key;
   private V val;
   private Node left, right;
   private int N;
   public Node(K k, V v, int n){
      key = k;
      val = v;
      N = n;
   }
}
```



- `K` and `V` are generic types; `K` is `Comparable`.
- The instance variable `N` gives the node count in the subtree rooted at the node.

BST Implementation (skeleton)

```
public class BST<K extends Comparable<K>, V>{
    private Node root; // root of BST
    private class Node{
        /* see previous slide . */
    }
    public V get(K k){
        /* see next slides */
    }
    public void put(K k, V v){
        /* see next slides */
    }
    public void delete(K k){
        /* see next slides */
    }
    public Iterable <K> keys(){
        /* see next slides */
    }
}
```

BST Search

- `get(key)`: Return value corresponding to given key, or `null` if no such key.
  - if the tree is empty, we have a *search miss*
  - if the search key is equal to the key at the root, we have a *search hit*.
  - Otherwise, search (recursively) in the appropriate subtree, moving left if key is smaller, right if it is larger.

Recursive form

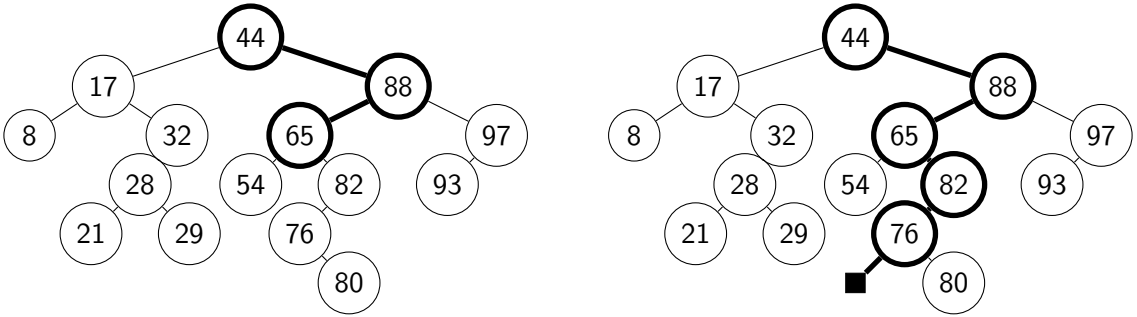
```
public V get(K key){
   return get(root, key);
}
private V get(Node x, K key){
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
}
```

Nonrecursive form

```
public Value get(K key){
    Node x = root;
    while (x != null){
       int cmp = key.compareTo(x.key);
       if(cmp < 0) x = x.left;
       else if (cmp > 0) x = x.right;
       else if (cmp == 0) return x.val;
   }
   return null;
}
```

- Cost: $1 +$ depth of tree $= O(h)$

BST Search Example



(a) A successful search (search hit) for key 65 in a binary search tree; (b) an unsuccessful search (search miss) for key 68 that terminates at the leaf to the left of the key 76.

BST Insert (put(key, val))

- Search for the key:
  - if the tree is empty, return a new node containing the key and value;
  - if the search key is less than the key at the root, set the left link to the result of inserting the key into the left subtree;
  - otherwise, set the right link to the result of inserting the key into the right subtree.

```
public void put(K key, V val){
   root = put(root, key, val);
}
private Node put(Node x, K key, V val){
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if (cmp < 0) x.left = put(x.left, key, val);
   else if (cmp > 0) x.right = put(x.right, key,
         val);
   else x.val = val;
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

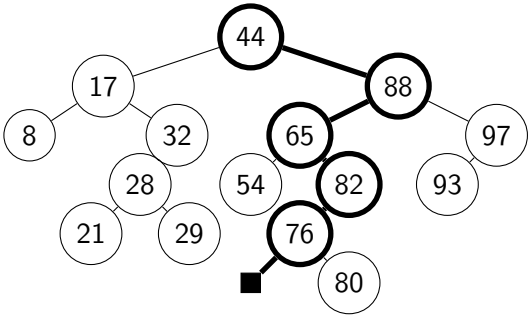
Node counts:

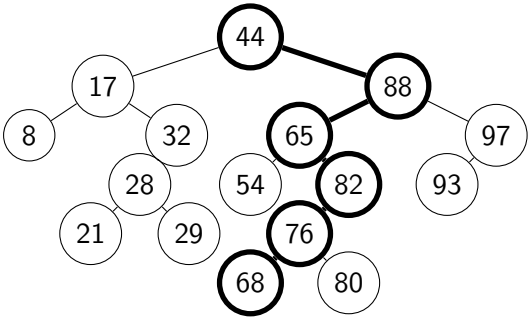
```
public int size(){
    return size(root);
}
private int size(Node x){
    if(x == null) return 0;
    else return x.N;
}
```

The `private size()` assigns the value 0 to `null` links, so that we can make sure that the invariant:
`size(x) = size(x.left) + size(x.right) + 1`
holds for every node x in the tree.

Cost: $1 +$ depth of tree $= O(h)$

BST Insert Example





Insertion of an entry with key 68 into the search tree. Finding the position to insert is shown in (a), and the resulting tree is shown in (b)

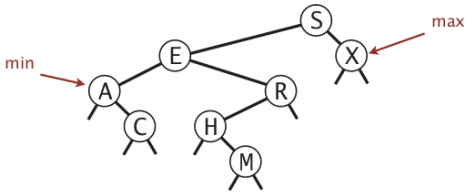
Inorder Traversal in BST

Inorder traversal of a BST yields keys in ascending order.

```
public Iterable <Key> keys(){
    Queue<K> q = new LinkedList<K>();
    inorder(root, q);
    return q;
}
private void inorder(Node x, Queue<K> q){
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);
    inorder(x.right, q);
}
```

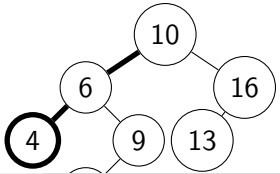
Minimum and Maximum

- Minimum (smallest key in the tree) is found in the leftmost node which does not have a left child.
- Maximum (largest key in the tree) is found in the right most node which does not have right child.



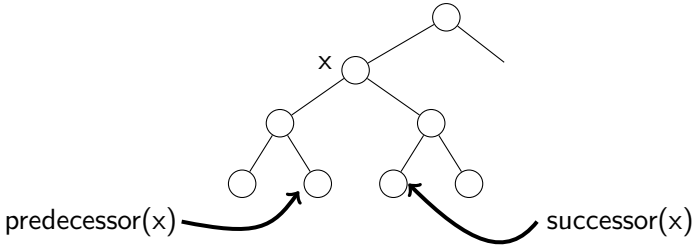
Algorithm for finding the minimum key.

```
public K min(){
    return min(root).key;
}
private Node min(Node x){
    if (x.left == null) return x;
```



Successor and Predeccessor

- The successor of a node `x` is the node with the smallest key greater than `x.key`.
- The predecessor of a node `x` is the node with the largest key lest than `x.key`.
- If `x` has two children then its predecessor is the maximum value in its left subtree `max(x.left)` and its successor is the minimum value in its right subtree `min(x.right)`.



- If `x` does not have a left child, then its predecessor is its first left ancestor.
- If `x` does not have a right child, then its successor is its first right ancestor.

Deleting the minimum/maximum

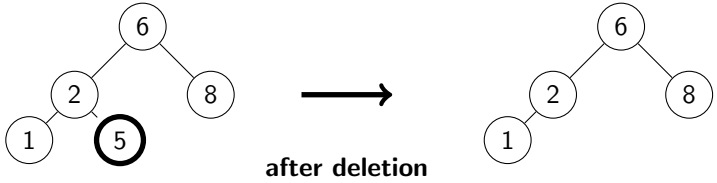
- To remove the key-value pair with the smallest key:
  - go left until you find a Node that has a `null` left link and then replace the link to that node by its right link

```
public void deleteMin(){
   root = deleteMin(root);
}
private Node deleteMin(Node x){
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   return x;
}
```

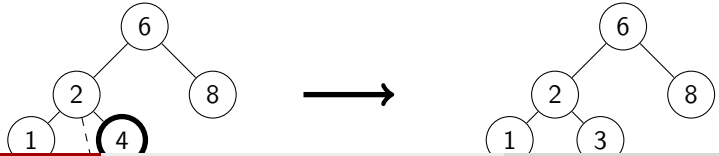
- The `deleteMax()` method is the same as `deleteMin()` with right and left interchanged.

Deleting from a BST (`delete(K key)`)

- More complicated than the other operations.
- To delete an item, first find the node with the key and consider the following cases:
  - **Case 1**: If the element to be deleted is a leaf node: return *NULL* to its parent.

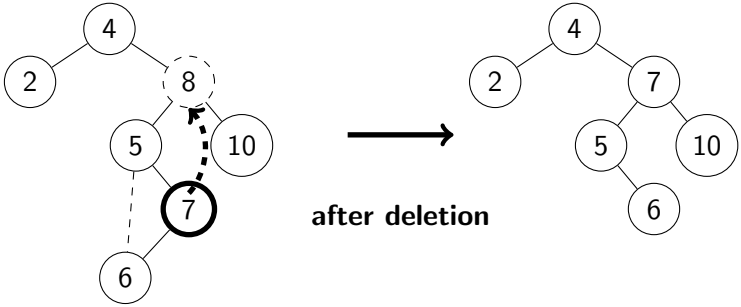


- **Case 2:** If the element to be deleted has one child, send the current node's child to its parent.



Deleting from a BST (`delete(K key)`)

- **Case 3**: If the element to be deleted has both children the general strategy is to replace the key of this node with the *largest element* of the left subtree and recursively delete that node.
  - The largest node in the left subtree cannot have a right child, so the second delete is an easy one.



Implementing delete(K key)

```
public void delete(K key){
   root = delete(root, key);
}
private Node delete(Node x, K key){
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if (cmp < 0) x.left = delete(x.left, key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else{
      if (x.right == null) return x.left; //case 1 or 2
      if (x.left == null) return x.right; //case 1 or 2

      // case 3
      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

Analysis of BSTs

- The dictionary operations `get`, `put` and `delete` have the worst-case running $O(h)$ where h is height of the tree.
- In the best case, a binary search tree has height $h = \lceil lg(n+1) \rceil - 1$.

Exercise

For the set of $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.