Java Collection Framework Implementation

(An introduction to data structures and algorithms)

An opportunity to present several important data structures for the efficient manipulation of collections and maps.

An opportunity to study the relative run-time complexity of different data structures and algorithms.

The study of data structures and algorithms is a central theme of Programming III.

Interface `List`

Interface `List<E>` is an extension of interface `Collection`.

A list is an indexed sequence of Objects.

Basic operations:

```
boolean isEmpty();
int size();

boolean contains(Object obj);
boolean add(E obj);
boolean remove(Object obj);

E get(int index);
Object set(int index, E obj);
void add(int index, E obj);
E remove(int index);
```

Implemented by classes `ArrayList` and `LinkedList`.

Class `ArrayList`

An implementation using simple, growing, possibly shrinking, arrays.

Class `ArrayList` (cont.)

```
public class ArrayList<E> implements List<E>{

    // Instance fields
    private int size = 0;       // number of elements in list
    private E[] elements;  // list = elements[0..size-1]

    // Constructors
    public ArrayList() {
        elements = (E[]) new Object[10];
    }

    public ArrayList(int capacity) {
        elements = (E[]) new Object[capacity];
    }

    public int size() {
        return size;
    }
```

Class `ArrayList` *(cont.)*

```
// Adds obj at end of list and returns true
public boolean add(E obj) {
    ensureCapacity(size + 1);
    elements[size++] = obj;
    return true;
}

// Adds obj at position index of list
public void add(int index, E obj) {
    ensureCapacity(size + 1);
    for (int i = size; i >= index+1; i--)
        elements[i] = elements[i-1];
    elements[index] = obj;
    size++;
}
```

Class `ArrayList` *(cont.)*

```
// Increase size of array to at least minCapacity
public void ensureCapacity(int minCapacity) {
    int oldCapacity = elements.length;
    if (minCapacity > oldCapacity) {
        int newCapacity =
             Math.max(minCapacity, 2*oldCapacity);
        E[] newElements = (E[])new Object[newCapacity];
        for (int i = 0; i < size; i++)
            newElements[i] = elements[i];
        elements = newElements;}
    }
}
```

Class `ArrayList` *(cont.)*

Removal methods require shifting sequences of elements one position to left.

Exercise Implement methods:

```
/**
 * Removes and returns the element at the specified position.
 * @param index the index of the element to removed.
 * @return the element previously at the specified position.
 */
 E remove(index)

/**
 * Removes the first occurrence of the specified element.
 * If this list does not contain the element, it is unchanged.
 * @param o - element to be removed from this list, if present.
 * @return true if this list contained the specified element.
 */
 boolean remove(obj)
```

Class `ArrayList` *(cont.)*

As an optimisation, when `size` becomes less than, say, `capacity/3`, we may copy the remaining elements into a shorter array of, say, `capacity/2` elements, and replace the longer array with the shorter array.

See the implementation, **`SimpleArrayList.java`**.

Note the implementation of interface `Iterator` over.

Complexity ... is complex. In summary, indexed access operations (`get`) are $O(1)$, i.e., constant time, whereas update operations (`add`, `remove`) are $O(n)$, i.e., linear time.

Class `ArrayList` (cont.)

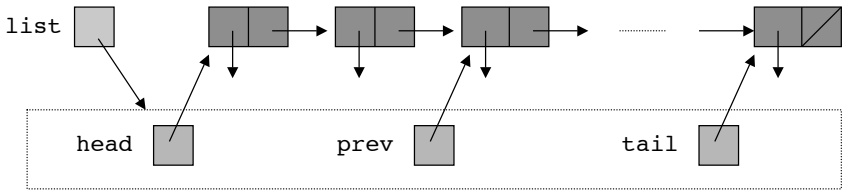
```
    /**
     * Class SimpleIterator provides a simple implementation
     * of Iterator based on SimpleArrayList.
     */
    private static class ArrayListIterator<E> implements
Iterator<E> {
        private List list;
        private E[] data;
        private int size;
        private int index;

        private ArrayListIterator(E[] data, int size,
                                  List<E> list) {
            this.list = list;
            this.data = data;
            this.size = size;
            this.index = 0;
        }

        public boolean hasNext() {
            return index < size;
        }
```

Class `ArrayList` (cont.)

```
        public E next() {
            if (index == size)
                throw new NoSuchElementException();

            Object value = data[index];
            index++;
            return value;
        }

    }// End class ArrayListIterator

    /**
     * Returns an iterator for the list.
     */
    public Iterator iterator() {
        return new ArrayListIterator(elements, size, this);
    }

} // End class ArrayList
```

Class `LinkedList` (L&O, Chapter 15)

An example of a linked data structure.



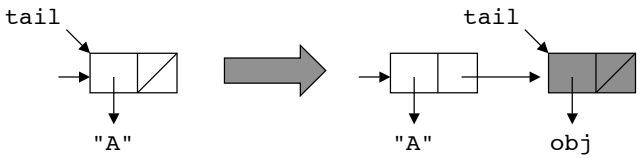

```
private class Node { // representation of a single node
    Object element;
    Node next;

    Node(Object element, Node next) {
        this.element = element;
        this.next = next;
    }
}
```

Class `LinkedList` (cont.)

```
public class LinkedList<E> implements List<E> {

    private Node head;  // first node of list
    private Node tail;  // last node of list
    private Node prev;  // previous node
    private int size;   // number of elements in list

    public LinkedList() {
        head = null;
        size = 0;
    }

    public int size() {
        return size;
    }
```

Class LinkedList (cont.)

```
// Adds obj at end of list and returns true
public boolean add(E obj) {
    if (head == null) {
        // Add obj to empty list
        head = new Node(obj, null);
        tail = head;
    } else {
        // Add obj at end of nonempty list
        tail.next = new Node(obj, null);
        tail = tail.next;
    }
    return true;
}
```

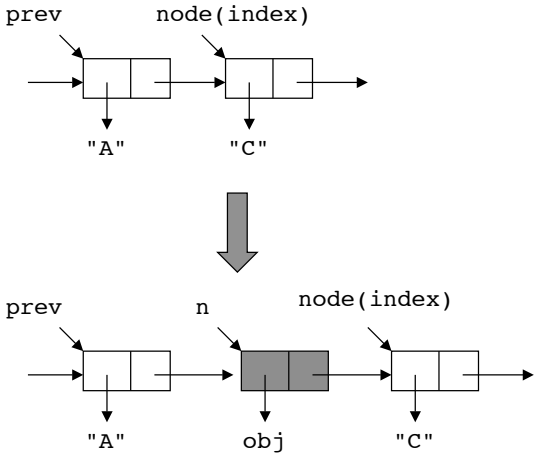


Class LinkedList (cont.)

```
// Adds obj at position index of list
public void add(int index, E obj) {
    Node n = new Node(obj, null);

    if (size == 0) {            // add to empty list
        head = n;
        tail = n;
    } else if (index == size) { // add after last node
        tail.next = n;
        tail = n;
    } else if (index == 0) {    // add before first node
        n.next = head;
        head = n;
    } else {                    // add before internal node
        n.next = node(index);
        prev.next = n;
    }
    size++;
}
```

Class LinkedList (cont.)

Add new node before position index:



Class `LinkedList` (cont.)

Method `node(index)` must step from start (`head`) of list, requiring *O(n)* steps.

```
// Returns node at position index of list,
// and sets prev to preceding node, if any
private Node node(int index) {
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException(
            "Index: " + index + ", Size: " + size);

    Node n = head;
    prev = null;
    for (int i = 0; i < index; i++) {
        prev = n;
        n = n.next;
    }
    return n;
}
```

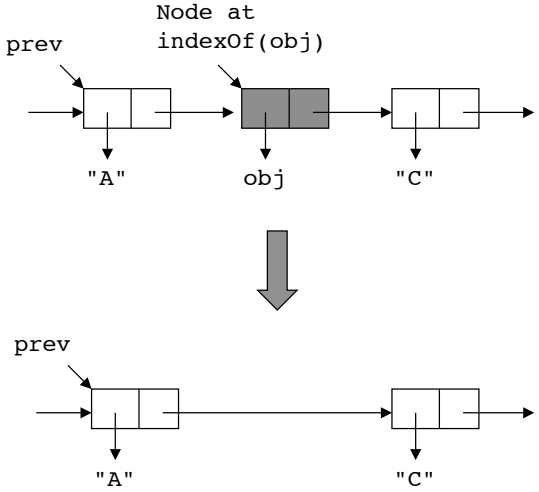
Class `LinkedList` *(cont.)*

Method `remove` requires a similar initial search to find the right position, using `node(index)` or `indexOf(obj)`, then the inverse of the `add` operation.

```
// Removes obj from list, and returns whether list changed
public boolean remove(E obj) {
    // prev will point to the node before E
    int index = indexOf(obj);
    if (index == -1) {
        return false;
    } else if (index == 0) { // remove first node
        head = head.next;
        return true;
    } else { // remove internal or last node
        prev.next = prev.next.next;
        return true;
    }
}
```

Class `LinkedList` *(cont.)*

Remove internal node:



Class `LinkedList` *(cont.)*

See the simplified implementation, **`SimpleLinkedList.java`**.

Note the implementation of interface `Iterator`.

Complexity Basically, update operations require *O*(1), i.e., constant time, whereas indexed operations require *O*(*n*), i.e., linear time.

Optimisations are possible. For example, we could store the index, `prevIndex`, of node `prev`, and search from node `prev` if the desired index is greater than or equal to `prevIndex`. This would allow iterations by increasing index to run in linear time, just as with `SimpleArrayList`.

Exercise Implement the method `node()` with this optimisation.

See the JDC tips under "Resources" on the Web page for a comparison of these two list implementations.

Interface `Map`

A map is a function or mapping from a set of distinct keys to a collection of corresponding values. (Keys and values are objects.)

Equivalently, a map is a set of (key, value) pairs, in which all keys are distinct.

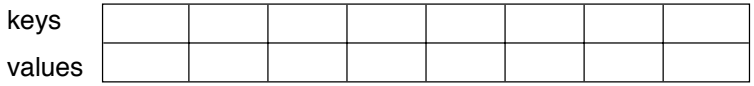
Basic operations:

```
Map<K,V>
boolean isEmpty();
int size();

boolean containsKey(Object key);
boolean containsValue(Object value);

V get(Object key);
V put(K key, V value);
V remove(Object key);
...
```

Implemented by classes `HashMap` and `TreeMap`.

Implementation of interface `Map`

The natural implementation is an array (or linked list) of (key, value) pairs.



Now what?

`Get` requires a linear search, or O(N) key comparisons..

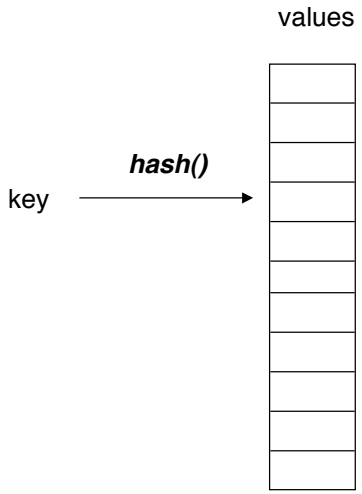
If keys are ordered, `get` can use binary search, which still requires O(log N) key comparisons.

Even if keys are ordered, `put` and `remove` require shifting all (key, value) pairs to right of insertion/removal position, requiring O(N) operations.

Surely we can do better! (We aim for constant-time operations.)

1. Hashing

Suppose we could index an array by keys rather than integers 0, 1, ..., N-1.



Hashing (cont.)

If all values are initially null, we have the following abstract implementations:

```
V put(K key, V value) {
    int index = hash(key);
    V oldValue = values[index];
    values[index] = value;
    return oldValue;
}

V get(Object key) {
    return values[hash(key)];
}

V remove(Object key) {
    int index = hash(key);
    Object oldValue = values[index];
    values[index] = null;
    return oldValue;
}
```

Hashing (cont.)

To make this work, the function hash must have the following properties.

- For each key, hash(key) is in the range 0 to N-1.
- If key1 != key2, then hash(key1) != hash(key2).
- Hash must use all the bits of key.

Hashing (cont.)

There are two big problems to solve:

1. How can we implement such a hash function?

2. What happens when two distinct keys are mapped to the same index? (Collision resolution) This will sometimes happen as there are many more possible keys than there are indexes.

Hash function implementation

1. Do it yourself

Let's restrict attention to strings (for simplicity).

We need to ensure that different strings return different values:

- hash(`"key1"`) != hash(`"key2"`) (use all characters)
- hash(`"abcd"`) != hash(`"dbca"`) (use each character differently)

We need the value to be in the range 0 to N-1.

```
int hash(String key) {
    int index = 0;
    for (int i = 0; i < key.length(); i++)
       index = 2*index + key.charAt(i);
    return (index & 0x7FFFFFFF) % N; // cf. Math.abs(index)
}
```

Hash function implementation (cont.)

2. Let Java do it for you

The class `Object` contains a method for this purpose:

```
int hashCode();
```

Method `hashCode()` returns a different integer for every object (as best as it can).

To use it as a hash function, we need to convert the integer to a nonnegative integer, and then to an integer in the range 0 to N-1:

```
int hash = key.hashCode();
int index = (hash & 0x7FFFFFFF) % N;
```

You can redefine `hashCode` (and `equals`) in user-defined classes, **provided** `key1.equals(key2)` implies `key1.hashCode() == key2.hashCode()`. You **should** redefine `hashCode` whenever you redefine `equals`.

Collision resolution

A collision occurs when two different keys are hashed to the same index.

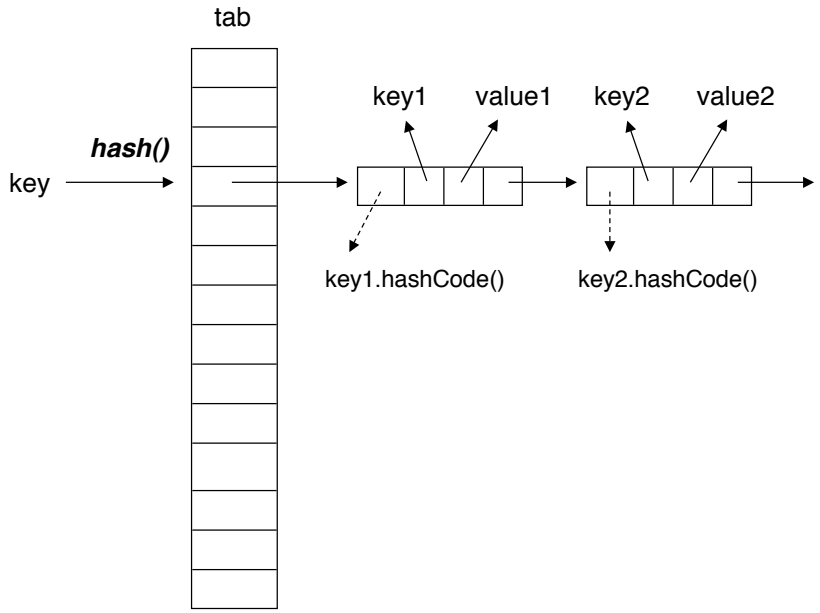
Many different solutions have been proposed. (Three are shown below.)

In every case, we must now store both keys **and** values in the table.

- Store the second (key, value) pair in the next free position in the table ("open addressing"). (Variants are possible.)
- Store the second (key, value) pair in a separate, "overflow" part of the table. (Variants are possible.)
- Store a list (or set) of all (key, value) pairs whose keys hash to the same table index.

We shall only consider the third of these solutions.

Collision resolution (cont.)



Collision resolution (cont.)

Each table element is a (singly-linked) list of (key, value) pairs whose keys hash to that index.

Each node (or entry) in these lists also contains the hash value of its key.

Abstract implementation of method `get`:

```
V get(Object key) {
    int hash = key.hashCode();
    int index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<k,V> e = tab[index]; e != null; e = e.next)
        if (key.equals(e.key))
            return e.value;  // key found
    return null;             // key not found
}
```

Collision resolution (cont.)

```
private static class Entry<K,V> implements Map.Entry<K,V> {
    int hash;
    K key;
    V value;
    Entry<K,V> next;
    // constructor
    Entry(int hash, K key, V value, Entry<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }
    // Map.Entry Ops
    public K getKey() {return key;}
    public V getValue() {return value;}
    public V setValue(V value) {
        Object oldValue = this.value;
        this.value = value;
        return oldValue;
    }
}
```

Collision resolution (cont.)

Abstract implementation of method `put`:

```
V put(K key, V value) {
    // Check whether the key is already in the table
    hash = key.hashCode();
    index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<K,V> e = tab[index]; e != null; e = e.next)
        if (key.equals(e.key)) { // key found, so change value
            V oldValue = e.value;
            e.value = value;
            return oldValue;
        }
    // Otherwise, create and add the new key-value entry
    Entry<K,V> e = new Entry<K,V>(hash, key, value,tab[index]);
    tab[index] = e;
    count++;
    return null;
}
```

Collision resolution (cont.)

To achieve constant-time operations, we need to keep the individual lists short. This is controlled by the following parameters:

`loadfactor` Ratio of number of (key, value) pairs to table elements. Need to keep this less than 0.75 (to avoid collisions, and to keep lists short).

`capacity` Number of table elements

When the total number of (key, value) pairs exceeds `capacity*loadfactor`, we need to allocate a larger table (cf. `ArrayList`), and rehash all (key, value) pairs to the new table.

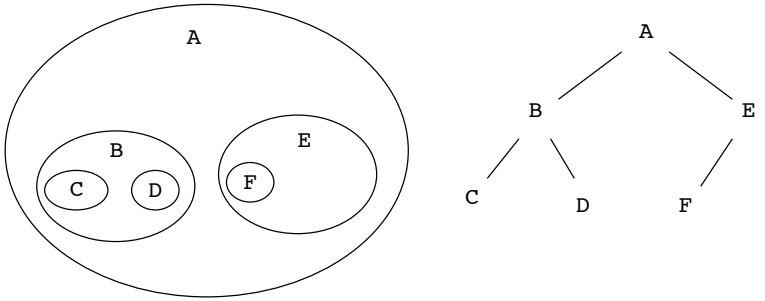
All operations are implemented in close to constant time (because lists are kept short).

See the simplified implementation **`SimpleHashMap.java`**.

2. Binary search trees

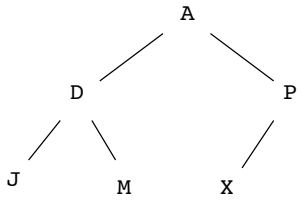
This alternative implementation approach preserves the natural order of keys at the cost of a slight reduction in performance.

Definition A **binary tree** is a set of nodes that is either empty or is partitioned into a **root** node and two subsets called the **left** subtree and the right subtree. Each subtree may itself have a root and two subtrees, and so on.

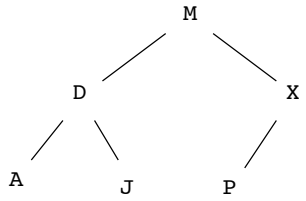


Binary search trees (cont.)

Definition A **binary search tree** is a binary tree in which all the nodes in the left subtree are less than the root node and all the nodes in the right subtree are greater than the root node, and similarly in each subtree (assuming the natural order on the values in the nodes).



Not a binary search tree



A binary search tree

Binary search trees (cont.)

The class `TreeMap` implements the interface `Map` as a binary search tree, in which each node contains a key and a value.

To find (get) a node with a given key, compare the key with the key at the root node. If they are equal, return the value at that node. If the given key is less than the key at the root , move to the left subtree (if any) and repeat. If the given key is greater than the key at the root, move to the right subtree (if any) and repeat.

To add (put) a node with a given key, find the position where such a node should occur in a similar way, and add the node as the root of a new left or right subtree. *e.g.*, add F as new left subtree of J.

To remove a node, a similar but slightly more complex operation is required.

All operations on a binary tree must preserve its defining property (all nodes in the left subtree must be less than the root node which must be less than all nodes in the right subtree, and similarly in every subtree).

Class `TreeMap`

First, a local class within class `TreeMap`:

```
/** Tree node. */
private static class Entry<k,V> implements Map.Entry<K,V> {
    K key;
    V value;
    Entry<K,V> left, right;

    Entry (K key, V value, Entry<K,V> left, Entry<K,V> right){
        this.key = key;
        this.value = value;
        this.left = left;
        this.right = right;
    }

   public K getKey() {return key;}
   // ...
}
```

Class `TreeMap` (cont.)

```
class TreeMap<K,V> implements Map<K,V> {

    private Entry<K,V> root;  // The root of the tree
    private int size;    // The number of pairs in the map

    /** Creates a new tree map. */
    public TreeMap() { root = null; size = 0; }

    /** Returns the number of pairs in the map. */
    public int size() { return size; }

    /** Returns whether or not the map is empty. */
    public int isEmpty() { return size == 0; }
```

Class `TreeMap` (cont.)

```
    /** Does the map contains a pair with the given key? */
    public boolean containsKey(Object key) {
        Entry<K,V> node = root;
        while (node != null) {
            int comparison = node.key.compareTo(key);
            if (comparison < 0) { // key in right subtree
                node = node.right;
            } else if (comparison == 0) { // key found
                return true;
            } else /*comparison > 0*/ { // key in left subtree
                node = node.left;
            }
        }
        return false;
    }
```

Class `TreeMap` (cont.)

```
    /** Returns the value associated with the given key. */
    public V get(Object key) {
        Entry<K,V> node = root;
        while (node != null) {
            int comparison = node.key.compareTo(key);
            if (comparison < 0) {
                node = node.right;
            } else if (comparison == 0) {
                return node.value;
            } else /*comparison > 0*/ {
                node = node.left;
            }
        }
        return null;
    }
```

Class `TreeMap` (cont.)

```
    /** Returns the value associated with the given key. */
    public V put(K key, V value) {
        Entry<K,V> node = root;
        Entry<K,V> prev = null; // parent of node
        int dirn;          // left or right subtree of parent

        // Find the node to extend or update
        while (node != null) {
            int comparison = node.key.compareTo(key);
            if (comparison < 0) {
                prev = node; dirn = RIGHT;
                node = node.right;
            } else if (comparison == 0) {
                break;
            } else /*comparison > 0*/ {
                prev = node; dirn = LEFT;
                node = node.left;
            }
        }
```

Class `TreeMap` (cont.)

```
        if (node == null)
            // Add a new node to the tree
            if (dirn == LEFT) {
               prev.left = new Node(key, value, null, null);
               return null;
            } else /* dirn == RIGHT */ {
               prev.right = new Node(key, value, null, null);
               return null;
            }
        else {
            // Update a node in the tree
            V oldValue = node.value;
            node.value = value;
            return oldValue;
        }
    }
```

Traversing `TreeMap`s

The nodes of a binary search tree can be listed recursively, in natural order, as follows:

```
public void traverse(Entry<K,V> t) {
    if (t != null) {
        if (t.left != null) traverse(t.left);
        "visit t";
        if (t.right != null) traverse(t.right);
    }
}
```

But this recursive method can't be used in a key set or entry set which require iterative traversal. For that we need a method `successor()` that returns the (natural, or **inorder**) successor of each node in a binary (search) tree. This can be used to implement the method `next()` in an iterator for a key set, value collection, or map entry set of a tree map. The first element of such an iterator is the *leftmost descendent* of the root of the tree. Subsequent elements are found using the method `successor()`.

Traversing `TreeMap`s (iteratively)

```
/** Returns inorder successor of node t in tree. */
private Entry<K,V> successor(Entry<K,V> t) {
    if (t == null)
        return null;
    else if (r.right != null) {
        // return leftmost descendent of right child
        Entry<K,V> p = t.right;
        while (p.left != null)
            p = p.left;
        return p;
    } else {
        // return closest ancestor of which t is
        // the rightmost descendent
        Entry<K,V> ch = t, p = t.parent,
        while (p != null && ch == p.right) {
            ch = p; p = p.parent;
        }
    return p;
}
```

Performance of class `TreeMap`

On average, a binary (search) tree with *N* nodes (key-value pairs) has a maximum path length (distance from root to leaf) of *log N* nodes. As all the key operations (`containsKey`, `add`, `get`, `put`, `remove`) perform at most one critical operation (a comparison) for each node on the path from root to leaf, they are all *O*(*log N*) operations, which is not as good as for `HashMap`, but still pretty good!

Interface `Set`

Interface `Set` is an extension of interface `Collection`.

It can conveniently be implemented as a `HashMap`, by concentrating on the keys (or elements), and ignoring the values.

Each element in the set is represented by a key with a non-null value (*e.g.,* `PRESENT`).

`Set` operations can then be implemented directly using `Map` operations. For example, a set contains an element `obj` if and only if the map corresponding to the set contains `obj` as a key; adding an element `obj` to a set is equivalent to adding the pair `(obj, PRESENT)` to the corresponding map.

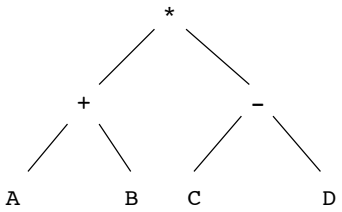
See the simplified implementation **`SimpleHashSet.java`**.

Exercise Extend the implementation by defining the method `iterator()`, that returns an iterator for the set.

Exercise Implement **`SimpleTreeSetjava`**.

Binary trees

Binary trees occur frequently in programming applications, *e.g.*, in expressions such as `(A+B)*(C-D)`:



We may traverse binary trees in depth-first or breadth-first fashion. With depth-first, there are three options:

- Preorder: Visit root, then traverse left subtree in preorder, then traverse right subtree in preorder: `* + A B - C D`
- Inorder: Traverse left subtree in preorder, then visit root, then traverse right subtree in inorder: `A + B * C - D`
- Postorder: Traverse left subtree in postorder, then traverse right subtree in postorder, then visit root: `A B + C - D *`

Binary trees (cont.)

Depth-first traversal is most easily implemented using recursion:

```
void preorder(Node t) {
    Visit t;
    if (t.left != null) preorder(t.left);
    if (t.right != null) preorder(t.right);
}
```

Breadth-first search (`* + - A B C D`) requires a first-in last-out queue, implemented using a list:

```
void breadthFirst(Node t) {
    List nodes = new ArrayList();
    nodes.add(t);
    while (! nodes.isEmpty()) {
        Node next = nodes.remove(0);
        Visit next;
        if (next.left != null) nodes.add(next.left);
        if (next.right != null) nodes.add(next.right);
    }
}
```