EECS 2011 Notes

Richard Robinson

January 1, 2019

Contents

1 Fundamental Data Structures

1.1 Insertion-Sort Algorithm time $\mathcal{O}(n^2)$

The insertion-sort algorithm is used to sort an array of elements. Analytically, the non-recursive version of algorithm operates by:

1. Iterate through each element e in the array
2. For each element, compare to each element left of e.
3. If e is less than such element, swap the indices of those elements

because the algorithm fully iterates through 2 nested loops, the complexity is described by $\mathcal{O}(n^2)$. Expressed algorithmically, it is represented as:

```
insertionSort(arr, arr.size() - 1);
public static void insertionSort(List<T> arr, int n) {
    if (n > 0) {
        insertionSort(arr, n - 1);
        int j;
        T elem = arr.get(n);

        for (j = n - 1; j >= 0 && arr.get(j) > elem; j--) {
            arr.set(j + 1, arr.get(j));
        }
        arr.set(j + 1, elem)
    }
}
```

The recursive version shown above improves upon this by

- turning the outer loop into a recursive call of size $n - 1$, and
- separating the swapping algorithm from the innermost loop

1.2 Singly Linked Lists time $\mathcal{O}(n)$

A linked list is a collection of nodes that form a linear sequence. Each node stores a reference to an element of the sequence, as well as a reference to the next node and alternatively a reference to the previous node (doubly-linked list).

A linked list must keep a reference to the first element (the head) and the last element (the tail), which has `null` as its reference. An element may be added to the head and tail of a linked list, respectively, via:

```java
public void addFirst(T e) {
    Node newest =
        new Node(e, this.head);
    this.head = newest;
    size++;
}
```

```java
public void addLast(T e) {
    Node newest = new Node(e, null);
    this.tail.setNext(newest);
    this.tail = newest;
    size++;
}
```

To remove an element from the head of the list, `this.head` is simply dereferenced and assigned to `this.head.getNext()`

1.3 Other Linked Lists time $\mathcal{O}(n)$

In a ***circularly linked list*** (CCL), the reference of the tail node is linked to the head node instead of `null`. This requires an additional `rotate()` method, which assigns `tail` to the next element. A CCL has the following methods:

```java
public void addFirst(T e) {
    if (this.isEmpty()) {
        this.tail = new Node(e, null);
        this.tail.setNext(this.tail);
    } else {
        Node newest =
            new Node(e, tail.getNext);
        tail.setNext(newest);
    }
    size++
}
.
.
.
```

```java
public void removeFirst(T e) {
    if (this.isEmpty()) return null;
    Node head = tail.getNext();

    if (head == tail) this.tail = null
    else tail.setNext(head.getNext());
    size--;
}
```

```java
public void addLast(T e) {
    this.addFirst(e);
    this.rotate();
}
```

The primary use of circularly linked lists is in implementations of ***round-robin scheduling***, which for a list C, gives a time slice to `C.first()`, then executes `C.rotate()`.

2 Algorithmic Analysis

2.1 Notation and Definitions

The set of ***primitive operations*** consists of the following operations:

- Assigning a value to a variable or reference
- Performing an arithmetic operation or boolean comparison
- Accessing a single element of an array
- Calling or returning from a method

The main functions by which complexities are measured with reference to include:

$$\overbrace{c \qquad \log n \qquad n \qquad n\log n \qquad n^2}^{f(n)}$$

in order of increasing time complexity. The ***Big-Oh*** notation is defined as follows:

$$f(n) \in \mathcal{O}(g(n)) \iff \exists(c > 0)\exists(n_0 > 1)\forall(n \geq n_0)(f(n) \leq c \cdot g(n)) \tag{1}$$

2.2 Binary Search time $\mathcal{O}(\log n)$

Binary search is a recursive search algorithm for sorted arrays to see if it contains an element e. In this algorithm, a target value $v = \lfloor(\min + \max)/2\rfloor$ is selected. If $e > v$, the procedure is continued with $\min \mapsto v + 1$ and similarly for $e < v$, and continues until e is found or $\min \geq \max$.

Algorithmically, binary search is given by:

```
bSearch(List<T> data, data.size() / 2, 0, data.size() - 1);
static boolean bSearch(List<T> data, T target, T lo, T hi) {
    if (lo > hi) return false;
    T mid = (lo + hi) / 2;

    if (target == data.get(mid)) return true;
    return (target < data.get(mid))
        ? bSearch(data, target, lo, mid - 1);
        : bSearch(data, target, mid + 1, hi);
}
```

In general, a binary-based recursive algorithm tests base cases, and otherwise returns itself with one or more of two sets of parameters, based on a boolean comparison.

3 Stacks, Queues, & Deques

3.1 Stacks time $\mathcal{O}(1)$

A stack is an ***abstract data type*** (ADT) whose elements operate according to the ***last-in, first-out*** (LIFO) principle. A general stack API has the following methods:

`void push(T e)` Adds the element e to the top of the stack

`T pop()` Removes and returns the top element

`T peek()` Returns the top element of the stack, or `null` if empty.

Most commonly, stacks are used when a history or matching of elements in a list is required. In general, the following algorithm can be used to check if a string has matched delimiters:

```java
static boolean isMatched(String str, String start, String end) {
    Stack<String> st = new Stack<>();
    for (char c : str.toCharArray()) {
        if (start.contains(c)) st.push(c);
        else if (end.contains(c)) {
            if (buf.isEmpty()) return false;

            char p = buf.pop();
            if (end.indexOf(c) != start.indexOf(p)) return false;
        }
    }

    return buf.isEmpty();
}
```

3.2 Queues time $\mathcal{O}(1)$

The queue ADT operates on a FIFO principle such that the oldest element is retrieved first, with the following API methods:

`void enqueue(T e)` Adds the element e to the back of queue

`T dequeue()` Removes and returns the first element

Since there is no reference to the back of the queue, the formula

$$i_b = (i_f + n) \mod \ell_{\text{arr}} \tag{2}$$

in which i_f is the index of the first element, n is the number of elements, and ℓ is the array length.

3.3 Deques time $\mathcal{O}(1)$

A double-ended queue is an ADT of a generalization between queues and stacks, in that an element may be added to the front of the queue and/or removed from the back. A deque has these API methods:

`void addFirst(T e)` Adds the element e to the front of the deque (similarly, `addLast()`)

`T removeFirst()` Removes and returns the first element (similarly, `removeLast()`)

Typically, deques are implemented via doubly-linked lists.

4 Lists & Iterators

4.1 Positional Lists time $\mathcal{O}(1)$

Positional list ADTs use a ***cursor*** to define a particular position within a list without indices via a `getElement()` method. A positional list may implement a traversal method via:

```
public void traverse(PositinalList<T> pl) {
    Position<T> cursor;
    for (cursor = pl.first(); cursor != null; cursor = pl.after(cursor))
        // action for each element
}
```

Note, all positional list methods except `set(p,e)` and `remove(p)` return the position of the element.

4.2 Iterators

An iterator is a pattern for the abstraction of sequences, which is generally an interface with methods `E next()` and `boolean hasNext()`.

As well, there exists an `Iterator iterator()` for use in data structures, which returns an iterator of the elements in the collection, which is used as a substitute for the for each loop when `remove()` is used, as for example to filer a list:

```
public void filter(List<E> list, double threshold) {
    Iterator<E> iter = list.iterator();
    while (iter.hasNext()) if (iter.next() > threshold) {
        iter.remove();
    }
}
```

Thus, a class may implement a custom Iterator by creating a private inner class which implements Iterator and contains its subsequent methods, along with the following method in the outer class:

```
public Iterator<E> iterator() {
    return new customIterator();
}
```

where `customIterator` is the name of the inner class.

To support a for each type loop for positions of a positional list, the following nested Iterator and Iterable classes must be created:

```java
private class PosIterator implements Iterator<Position<E>> {
    private Position<E> cursor = first(), recent = null;
    public boolean hasNext() { return cursor != null; }

    public Position<E> next() {
        recent = cursor;
        cursor = after(cursor);
        return recent;
    }

    public void remove() {
        LinkedPosList.this.remove(recent);
        recent = null;
    }
}

private class PosIterable implements Iterable<Position<E>> {
    public Iterator<Position<E>> iterator() {
        return new PosIterator();
    }
}

public Iterable<Position<E>> positions() {
    return new PosIterable();
}

public class ElemIterator implements Iterator<E> {
    Iterator<Position<E>> posIterator = new PositionIterator();

    public boolean hasNext() { return posIterator.hasNext(); }
    public E next() { return posIterator.next().getElement(); }
    public void remove() { posIterator.remove(); }
}

public Iterator<E> iterator() { return new ElemIterator(); }
```