Data Structure Preview

Charles Zhang

Winter Break 2020

Contents

1 Time Complexity

1.1 Big-O Notation

Time Complexity of algorithms: T(N) = $O(f(\text{N}))$.

1.1.1 Definitions

Definition 1.1. *If $\exists$ c, $n_0 \in \mathbb{R}$ such that T(N) $\leq cf$(N) when $N \geq n_0$, then T(N) = O(f(N)).*

Definition 1.2. *If $\exists$ c, $n_0 \in \mathbb{R}$ such that T(N) $\geq cg$(N) when $N \geq n_0$, then T(N) = Ω(g(N)).*

Definition 1.3. *T(N) = Θ(h(N)) $\Leftrightarrow$ T(N) = O(h(N)) and T(N) = Ω(h(N)).*

Definition 1.4. *T(N) = o(p(N)) if T(N) = O(p(N)) and T(N) $\neq \Theta$(p(N)).*

T(N) = 1000N, f(N) = N^2
$\because \exists\ n_0 = 1000$, c = 1, T(N) $\leq$ cf(N) when N $\geq n_0$
$\therefore$ 1000N =$O(N^2)$.

1.1.2 In terms of relative rate of growth

the relative rate of growth of $T(N) \leq$ relative rate of growth of $f(n)$, T(N) = O(f(N));
the relative rate of growth of $T(N) \geq$ relative rate of growth of $g(n)$, T(N) = $\Omega(g(\text{N}))$;
the relative rate of growth of $T(N)$ = relative rate of growth of $h(n)$, T(N) = $\Theta(g(\text{N}))$;
the relative rate of growth of $T(N)$ ¡ relative rate of growth of $h(n)$, T(N) = $o(g(\text{N}))$

$2N^2 = O(N^4) = O(N^3) = O(N^2)$, but the last option is the best answer.

Function	Name
1	Constant
$logN$	Logarithmic
log^2N	Log-squared
N	Linear
$NlogN$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

1.1.3 Rules

Rule 1.1. *If $T_1(N) = O(f(n))$ and $T_2(N) = O(g(N))$, then*
(a) $T_1(N) + T_2(N) = O(f(N) + g(N))\ = O(max(f(N), g(N))$,
(b) $T_1(N) * T_2(N) = O(f(N) * g(N))$.

Rule 1.2. *If T(N) is a polynomial of degree k, then T(N) = $\Theta(N^k)$.*

Rule 1.3. *$log^kN = O(N)$ for any constant k(t logarithms grow very slowly).*

1.2 Running Time Calculations

Here is a simple program fragment to calculate $\sum_{i=1}^{N} i_3$.

```
public static int sum( int n ){
    int partialSum;
1   partialSum = 0;
2   for( int i = 1; i <= n; i++ )
3       partialSum += i * i * i;
4   return partialSum;
}
```

The declarations count for no time. Lines 1 and 4 count for one unit each. Line 3 counts for four units per time executed (two multiplications, one addition, and one assignment) and is executed N times, for a total of 4N units. Line 2 has the hidden costs of initializing i, testing i $\leq$ N, and incremented i. The total cost of all these is 1 to initialize, N + 1 for all the tests, and N for all the increments, which is 2N + 2. We ignore the costs of calling the method and returning, for a total of 6N + 4. **Thus, we say that this method is O(N).**

1.2.1 General Rules

Rule 1.4. - ***for loops***
The running time of a for loop is at most the running time of the statements inside the for loop (including tests) times the number of iterations.

Rule 1.5. - ***Nested loops***
Analyze these inside out. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

As an example, the following program fragment is $O(N^2)$*:*

```
for( i = 0; i < n; i++ )
    for( j = 0; j < n; j++ )
        k++;
```

Rule 1.6. - ***Consecutive Statements***
These just add.

As an example, the following program fragment, which has $O(N)$ *work followed by* $O(N^2)$ *work, is also* $O(N^2)$*:*

```
for( i = 0; i < n; i++ )
    a[ i ] = 0;
for( i = 0; i < n; i++ )
    for( j = 0; j < n; j++ )
        a[ i ] += a[ j ] + i + j;
```

Rule 1.7. - ***if/else***
For the fragment

```
if( condition )
    S1
else
    S2
```

the running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S1 and S2.

1.2.2 Conclusion

$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n))$

1.2.3 Logarithms in the Running Time

Binary Search: Given an integer X and integers A_0, A_1, ... , A_{N-1}, which are presorted and already in memory, find i such that A_i = X, or return i = -1 if X is not in the input.

```
/**
* Performs the standard binary search.
* @return index where item is found, or -1 if not found.
*/
public static <AnyType extends Comparable<? super AnyType>>
int binarySearch( AnyType [ ] a, AnyType x )
{
    int low = 0, high = a.length - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ].compareTo( x)<0)
            low = mid + 1;
        else if( a[ mid ].compareTo( x )>0)
            high = mid - 1;
        else
            return mid;    // Found
    }
    return NOT_FOUND; // NOT_FOUND is defined as -1
}
```

Clearly, all the work done inside the loop takes O(1) per iteration, so the analysis requires determining the number of times around the loop. The loop starts with $high - low$ = N - 1 and finishes with $high - low \geq$ -1. Every time through the loop the value $high - low$ must be at least halved from its previous value; thus, the number of times around the loop is at most [log(N - 1)] + 2(As an example, if $high - low = 128$, then the maximum values of $high - low$ after each iteration are 64, 32, 16, 8, 4, 2, 1, 0, -1). Thus, the running time is O(log N).
Binary search can be viewed as our first data structure implementation. It supports the contains operation in O(log N) time, but all other operations (in particular insert) require O(N) time.

2 Lists, Stacks, and Queues

2.1 Lists

General list of the form: A_0, A_1, A_2..., A_{N-1}. The size of this list is N.

2.1.1 Array Implementation of List

Linked List: To execute $printList$ or $find(x)$ we merely start at the first node in the list and then traverse the list by following the next links. This operation is clearly linear-time, as in the array implementation.

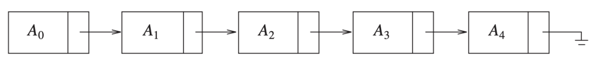

Figure 2.1: A linked list

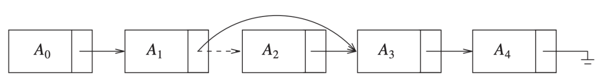

Figure 2.2: Deletion from a linked list

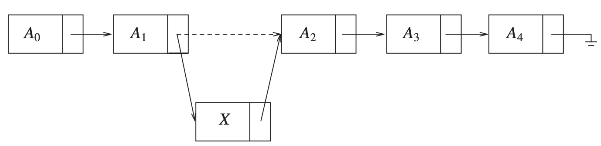

Figure 2.3: Insertion into a linked list

Codes Implementation

```java
import java.util.Arrays;

public class MyArray {

    // Array for store
    private int[] elements;

    public MyArray() {
        elements = new int[0];
    }

    // method for getting the length
    public int size() {
        return elements.length;
    }

    // add the element at the end of the array
    public void add(int element) {
        int[] newArr = new int[elements.length
             + 1];
        for (int i = 0; i < elements.length;
             i++) {
            newArr[i] = elements[i];
        }
        newArr[elements.length] = element;
        elements = newArr;
```

```java
}

// show the array
public void show() {
    System.out.println(Arrays.toString(elements));
}

// delete the element in the array
public void delete(int index) {
    // bound test
    if (index < 0 || index >
        elements.length - 1) {
        throw new RuntimeException("Out of
            Bounds!");
    }
    int[] newArr = new int[elements.length
        - 1];
    for (int i = 0; i < newArr.length; i++)
        {
        if (i < index) {
            newArr[i] = elements[i];
        } else {
            newArr[i] = elements[i + 1];
        }
    }
    elements = newArr;
}

// get the element
public int get(int index) {
    if (index < 0 || index >
        elements.length - 1) {
        throw new RuntimeException("Out of
            Bounds!");
    }
    return elements[index];
}

// insert element at the specific position
public void insert(int index, int element) {
    int[] newArr = new int[elements.length
        + 1];
    for (int i = 0; i < elements.length;
        i++) {
        if (i < index) {
            newArr[i] = elements[i];
        } else {
            newArr[i + 1] = elements[i];
        }
    }
    newArr[index] = element;
    elements = newArr;
}

// replace
public void set(int index, int element) {
    if (index < 0 || index >
        elements.length - 1) {
        throw new RuntimeException("Out of
            Bounds!");
    }
    elements[index] = element;
}

// linearly search
public int search(int target) {
    for(int i=0;i<elements.length;i++) {
        if(elements[i]==target) {
            return i;
        }
    }
    return -1;
}

// binary search
public int binarySearch(int target) {
    int begin = 0;
    int end = elements.length-1;
    int mid = (begin+end)/2;
    while(true) {
        if(begin>=end) {
            return -1;
        }
        if(elements[mid]==target) {
            return mid;
        }else{
            if(elements[mid]>target) {
                end=mid-1;
            }else {
                begin = mid+1;
            }
            mid=(begin+end)/2;
        }
    }
}
}
```

2.1.2 Node

- Create Nodes

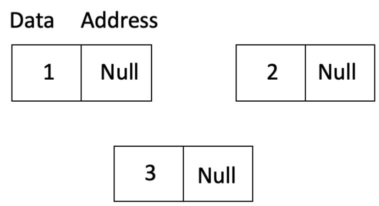

Figure 2.4: Create three Nodes with data but without address

```
public class Node {
    // Data in Node
    int data;
    // the next node
    Node next;

    public Node(int data) {
        this.data=data;
    }

public static void main(String[] args) {
        // Create Nodes
        Node n1 = new Node(1);
        Node n2 = new Node(2);
        Node n3 = new Node(3);
```

- Append Nodes

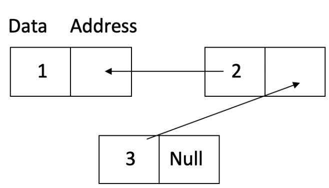

Figure 2.5: Append a node

```java
// Append Nodes
public Node append(Node node) {
    // Current
    Node currentNode = this;
    while(true) {
        // get out
        Node nextNode = currentNode.next;
        // last node is null
        if(nextNode==null) {
            break;
        }
        // assignment
        currentNode = nextNode;
    }
    // append the node to the next of the current node that need to be appended
    currentNode.next=node;
    return this;
}
```

2.2 Stacks

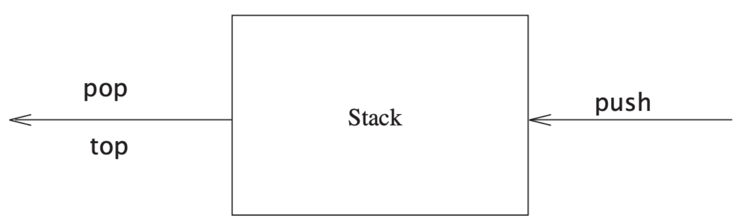

Figure 2.6: Stack model: input to a stack is by push, output is by pop and top

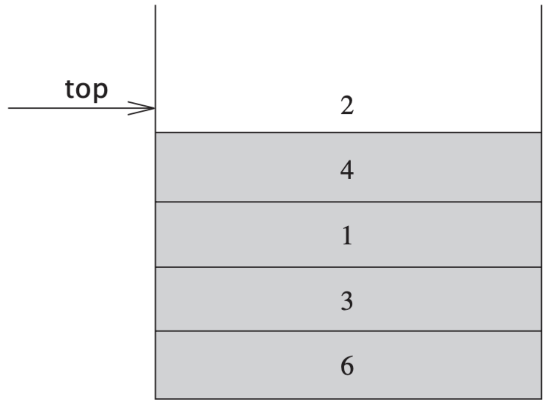

Figure 2.7: Stack model: Only the top element is accessible

2.3 Queues