CSC3100: Data Structure
Midterm Review

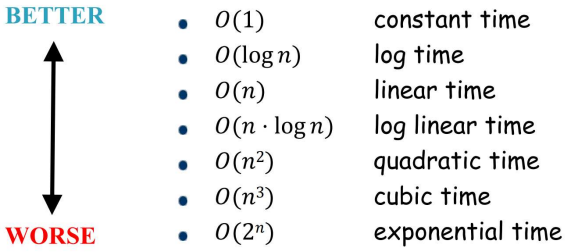
Baicheng Chen
SDS, CUHK-Shenzhen
baichengchen@link.cuhk.edu.cn

Array

1. Pro & Con of arrays:
   (1) Pro: If you know the index, you can find the object with one basic operation; Efficient search if array is sorted.
   (2) Con: The capacity is fixed, so if you insert an object in a place between two objects in an array, you have to move the object first.
2. An array is a linear data structure that hosts a collection of data elements stored at consecutive locations in a computer's memory.
3. Methods: createArray(n), retrieve(arr, i), store(arr, i, itemToStore)
4. Pass-by-Value vs. Pass-by-Reference

Time Complexity

1. Big-Oh Notation:
   Definition: $g(n) = O(f(n))$ if and only if there exist positive constants c and n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$.
   Properties:
   (1) $g_1(n) = O(f_1(n))$, $g_2(n) = O(f_2(n))$.
   Then, $g_1(n) \cdot g_2(n) = O(f_1(n) \cdot f_2(n))$.
   (2) Sum: The bigger of the two big.
   (3) Log only beats constant.
   (4) Exponential beats powers.



2. Big-Omega Notation:
   Definition: $g(n) = \Omega(f(n))$ if and only if there exist positive constants c and n_0 such that $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$. Reverse of Big-Oh.
   Remark: Big-Oh and Big-Omega are both not tight.
3. Big-Theta Notation:
   Definition: If $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$, then $g(n) = \Theta(f(n))$.
4. Master Theorem (Big-Oh version):

   Available: $T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d)$

$$T(n) = \begin{cases} O(n^d \log n), & if\ a = b^d \\ O(n^d), & if\ a < b^d \\ O\left(n^{\log_b a}\right), & if\ a > b^d \end{cases}$$

Sorting Algorithms

1. Outline:

Sorting algorithm	Stability	Time cost			Extra space cost
		Best	Average	Worst	
Bubble sort	√	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	√	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection sort	×	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
MergeSort	√	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
HeapSort	×	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
QuickSort	×	O(nlogn)	O(nlogn)	$O(n^2)$	O(1)
ShellSort	×	O(n)	$O(n^{1.3})$	$O(n^2)$	O(1)
CountingSort	√	O(n+k)	O(n+k)	O(n+k)	O(k)
BucketSort	√	O(n)	O(n+k)	$O(n^2)$	O(k)
RadixSort	√	O(nk)	O(nk)	O(nk)	O(n)

2. Counting Sort, Bucket Sort and Radix Sort don't need any comparisons.
3. Bubble sort: 0~i 范围上，相邻位置较大的数滚下去，最大值最终来到 i 位置，然后 0~i-1 范围上继续；Insertion sort: 0~i 范围上已经有序，新来的数从右到左滑到不再小的位置插入，然后继续；Selection sort：i~n-1 范围上，找到最小值并放在 i 位置，然后 i+1~n-1 范围上继续。
4. Quick sort:
   (1) Deterministic vs. randomized
   (2) Random(x) → partition(l, r, x) → quicksort(l, mid-1) → quicksort(mid+1, r)
   (3) Partition: Two pointer i, a. If [i] <= x, swap([i], [a]), i ++, a ++ ; If [i] > x, i ++ .
   (4) Merge sort vs quicksort:
   - Both MergeSort and QuickSort use the divide-and-conquer paradigm
   - When MergeSort executes the merge operation
     `Requires an additional array to do the merge operation
     `Needs to do additional data copy: copy to additional array and then

copy back to the input array
- When QuickSort executes the partition operation:
  `Operations on the same array
  `No additional space required
- QuickSort is typically 2-3 times faster then MergeSort

5. ShellSort: break the quadratic time barrier by comparing elements that are distant.
6. RadixSort: If m is large, countingsort is costly in both time and space.
   (1) Apply BucketSort on each digit (from least significant digit to most significant digit)

   (2) 所有桶里先进数字先出(Queue)

   (3) Since RadixSort is faster than QuickSort, why is QuickSort still preferable in many cases?
   - Although RadixSort runs in $\Theta(n)$ while QuickSort $\Theta(n \log n)$, QuickSort has a much smaller constant factor c
   - RadixSort requires extra memory, whereas QuickSort works in place.

List

1. Operations: printList, makeEmpty, Find, insert, delete, next&previous
2. Why list?
   (1) The limitations of array:
   - The size of the array is fixed, so we must know the upper limit on the number of elements in advance.
   - Inserting a new element in an array is expensive because the room has to be created for the new elements and existing elements have to be shifted.

   (2) Advantages:
   - Dynamic data structure: The size of a linked list is not fixed as it can vary arbitrarily.
   - Insertion and deletion are easier: In linked lists, insertion and deletion are easier than those in arrays, since the elements of an array are stored in a consecutive location, while the elements of a linked list are stored in a random location.
     If we want to insert or delete the element in an array, then we need to shift the elements for creating the space, while in a linked list, we do not have to shift the elements.
   - Memory efficient: Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.

   (3) Disadvantages:
   - Memory usage: The node in a linked list occupies more memory than array as each node has one simple variable and one pointer variable that occupies 4 bytes in memory.
   - Traversal: In an array, we can randomly access the element by index,

while in a linked list, the traversal is not easy, i.e., if we want to access the element in a linked list, we cannot access the element randomly.

- Reverse traversing: In a linked list, backtracking or reverse traversing is difficult.
  In a doubly linked list, it is easier but requires more memory.

3. Types of linked list: Singly linked list, doubly linked list, circular linked list, doubly circular linked list
4. Operations' time complexities:
   Insert & Delete: O(n) (worst). If give the pre_node.nest: O(1).
5. Reverse Linked List:

```
public static ListNode reverseList(ListNode head) {
    ListNode pre = null;
    ListNode next = null;
    while (head != null) {
        next = head.next;
        head.next = pre;
        pre = head;
        head = next;
    }
    return pre;
}
```

How about the double linked list?

```
public static DoubleListNode reverseDoubleList(DoubleListNode head) {
    DoubleListNode pre = null;
    DoubleListNode next = null;
    while (head != null) {
        next = head.next;
        head.next = pre;
        head.last = next;
        pre = head;
        head = next;
    }
    return pre;
}
```

6. Combine two linked lists:

```
public static ListNode mergeTwoLists(ListNode head1, ListNode head2) {
    if (head1 == null || head2 == null) {
        return head1 == null ? head2 : head1;
    }
    ListNode head = head1.val <= head2.val ? head1 : head2;
    ListNode cur1 = head.next;
    ListNode cur2 = head == head1 ? head2 : head1;
    ListNode pre = head;
    while (cur1 != null && cur2 != null) {
        if (cur1.val <= cur2.val) {
            pre.next = cur1;
            cur1 = cur1.next;
        } else {
            pre.next = cur2;
            cur2 = cur2.next;
        }
        pre = pre.next;
    }
    pre.next = cur1 != null ? cur1 : cur2;
    return head;
}
```

7. Partition of the linked list, for key value x, left < x, right >= x.

```
public static ListNode partition(ListNode head, int x) {
        ListNode leftHead = null, leftTail = null; // < x的区域
        ListNode rightHead = null, rightTail = null; // >=x的区域
        ListNode next = null;
        while (head != null) {
                next = head.next;
                head.next = null;
                if (head.val < x) {
                        if (leftHead == null) {
                                leftHead = head;
                        } else {
                                leftTail.next = head;
                        }
                        leftTail = head;
                } else {
                        if (rightHead == null) {
                                rightHead = head;
                        } else {
                                rightTail.next = head;
                        }
                        rightTail = head;
                }
                head = next;
        }
        if (leftHead == null) {
                return rightHead;
        }
        // < x的区域有内容!
        leftTail.next = rightHead;
        return leftHead;
}
```

```
public static boolean isPalindrome(ListNode head) {
    if (head == null || head.next == null) {
        return true;
    }
    ListNode slow = head, fast = head;

    while (fast.next != null && fast.next.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    ListNode pre = slow;
    ListNode cur = pre.next;
    ListNode next = null;
    pre.next = null;
    while (cur != null) {
        next = cur.next;
        cur.next = pre;
        pre = cur;
        cur = next;
    }

    boolean ans = true;
    ListNode left = head;
    ListNode right = pre;

    while (left != null && right != null) {
        if (left.val != right.val) {
            ans = false;
            break;
        }
        left = left.next;
        right = right.next;
    }

    cur = pre.next;
    pre.next = null;
    next = null;
    while (cur != null) {
        next = cur.next;
        cur.next = pre;
        pre = cur;
        cur = next;
    }
    return ans;
}
```

8. Determine whether the linked list is a palindrome structure: Use stack or fast-slow pointers (find the midpoint).
9. Return the first node goes into circle:
   (1) Use set and just traverse, if the node is in the set, return.
   (2) Fast-slow pointers: If fast pointer goes to null, no circle. If there exist circle, they must meet together.
   First, let them meet. Then, let the fast pointer return to the head, change the faster pointer to 1 step each time, the slower pointer keep 1 step. Finally, they must meet at the first circle node.

```
public static ListNode detectCycle(ListNode head) {
        if (head == null || head.next == null || head.next.next == null) {
                return null;
        }
        ListNode slow = head.next;
        ListNode fast = head.next.next;
        while (slow != fast) {
                if (fast.next == null || fast.next.next == null) {
                        return null;
                }
                slow = slow.next;
                fast = fast.next.next;
        }
        fast = head;
        while (slow != fast) {
                slow = slow.next;
                fast = fast.next;
        }
        return slow;
}
```

10. Matrix representation:
    (1) Triplet representation (minimum space). No!
        - Inefficient access
        - Increased complexity for row or column operations
    (2) Linked list representation

Stack

1. Operations: push(e), pop(), makeEmpty(), top(), isEmpty()
2. LIFO (Last-In-First-Out)
3. Implementations: Use Linked list and Array
    (1) Using array is faster: Memory allocation, continuous memory can be loaded into cache
    (2) Using list save space.
4. Postfix:
    How to derive the postfix? Each operator appears after its operands.

    e.g. a / b – c + d * e – a * c → a b / c – d e * + a c * –

5. Use queue to implement stack: One queues. Pop from head and push to tail again. When a new value wants to in, record how many values (n) are in the queue, then, the new value in, do pop and push n times. O(n) for push, other O(1).
6. Add a function of getMin(), to get the minimum value in the stack using O(1): Add a new stack min.

```
class MinStack1 {
        public Stack<Integer> data;
        public Stack<Integer> min;

        public MinStack1() {
                data = new Stack<Integer>();
                min = new Stack<Integer>();
        }

        public void push(int val) {
                data.push(val);
                if (min.isEmpty() || val <= min.peek()) {
                        min.push(val);
                } else { // !min.isEmpty() && val > min.peek()
                        min.push(min.peek());
                }
        }

        public void pop() {
                data.pop();
                min.pop();
        }

        public int top() {
                return data.peek();
        }

        public int getMin() {
                return min.peek();
        }
}
```

Queue

1. FIFO (First-In-First-Out)
2. Types: Linear Queue and its implementations, Circular queue, Priority queue, Double ended queue

3. Operations: enqueue(e) → Insert, dequeue() → remove

4. Linear Queue Implementations:
   (1) Linked List
   (2) Array: may run out of rooms.
   If the first several elements are deleted, we cannot insert more elements even though the space is available.
   Solutions: (1) Keep front always at 0 by shifting the contents up the queue, but this solution is inefficient. (2) Use a circular queue (wrap around & use a length variable to keep track of the queue length).

5. Priority Queue: it does not follow the FIFO principle → can be implemented by Array, Linked List, Binary Search Tree, Heap.
6. Double ended queue: It does not follow the FIFO principle. Can be used as stack and queue.
7. Use stack to implement queue: Two stacks (in, out). In for input, values from in to out, out for output. Every methods, O(1).

Tree and binary tree

1. Depth of a node: the length of unique path from the root to n, the root is at depth 0.
2. Height of a node: the length of the longest path from n to a leaf, all leaves are at height 0.
3. Binary tree:
   (1) Full binary tree: all the nodes have either two or no children.
   (2) Complete binary tree: all the leaves are completely filled except possibly the lowest one, which is filled from the left.

4. Operations: create(bintree), isEmpty(bintree), MakeBT(bintree1, element, bintree2)[left subtree is bintree1, right is 2], Lchild(bintree), Rchild(bintree), Data(bintree)[element data stored in the root node].
5. Design:
   (1) Using pointers: more intuitive
   (2) Using array: need more complicated design, and cannot efficiently handle all operations, will be used for heap.
   For any i-th node, parent(i) is $\left\lfloor \frac{i}{2} \right\rfloor$, leftChild(i) is at 2i if $2i \leq n$,
   rightChild(i) is at 2i+1 if $2i + 1 \leq n$.
   Pros and cons:
   Generalize to all binary trees, efficient for complete binary trees, but inefficient for skewed binary trees, inefficient to implement the ADT.
   从 1 开始以 full binary 的形式按行标号，没有出现的 node 要在 array 里跳过，留空。

6. Traversing strategies
   (1) Preorder (depth-first)
      Visit the node
      Traverse the left subtree
      Traverse the right subtree

   (2) Inorder → can be used to sort the keys of Binary Search Tree

      Left, node, right
   (3) Postorder
      Left, right, node
   (4) Reconstruction:
      Pre + In: OK
      Pre + Post: Impossible
      Post + In: OK
7. Traversing not using recursion: Using Stack!

```
public static void preOrder(TreeNode head) {
        if (head != null) {
                Stack<TreeNode> stack = new Stack<>();
                stack.push(head);
                while (!stack.isEmpty()) {
                        head = stack.pop();
                        System.out.print(head.val + " ");
                        if (head.right != null) {
                                stack.push(head.right);
                        }
                        if (head.left != null) {
                                stack.push(head.left);
                        }
                }
                System.out.println();
        }
}
```

```
public static void inOrder(TreeNode head) {
        if (head != null) {
                Stack<TreeNode> stack = new Stack<>();
                while (!stack.isEmpty() || head != null) {
                        if (head != null) {
                                stack.push(head);
                                head = head.left;
                        } else {
                                head = stack.pop();
                                System.out.print(head.val + " ");
                                head = head.right;
                        }
                }
                System.out.println();
        }
}
```

```
public static void posOrderTwoStacks(TreeNode head) {
        if (head != null) {
                Stack<TreeNode> stack = new Stack<>();
                Stack<TreeNode> collect = new Stack<>();
                stack.push(head);
                while (!stack.isEmpty()) {
                        head = stack.pop();
                        collect.push(head);
                        if (head.left != null) {
                                stack.push(head.left);
                        }
                        if (head.right != null) {
                                stack.push(head.right);
                        }
                }
                while (!collect.isEmpty()) {
                        System.out.print(collect.pop().val + " ");
                }
                System.out.println();
        }
}
```

```
public static void posOrderOneStack(TreeNode h) {
        if (h != null) {
                Stack<TreeNode> stack = new Stack<>();
                stack.push(h);
                // 如果始终没有打印过节点，h就一直是头节点
                // 一旦打印过节点，h就变成打印节点
                // 之后h的含义 : 上一次打印的节点
                while (!stack.isEmpty()) {
                        TreeNode cur = stack.peek();
                        if (cur.left != null && h != cur.left && h != cur.right) {
                                // 有左树且左树没处理过
                                stack.push(cur.left);
                        } else if (cur.right != null && h != cur.right) {
                                // 有右树且右树没处理过
                                stack.push(cur.right);
                        } else {
                                // 左树、右树 没有 或者 都处理过了
                                System.out.print(cur.val + " ");
                                h = stack.pop();
                        }
                }
                System.out.println();
        }
}
```

8. Verify complete binary tree: BFS
   How about verifying the binary search tree (BST)?

```java
public static boolean isCompleteTree(TreeNode h) {
        if (h == null) {
                return true;
        }
        l = r = 0;
        queue[r++] = h;
        // 是否遇到过左右两个孩子不双全的节点
        boolean leaf = false;
        while (l < r) {
                h = queue[l++];
                if ((h.left == null && h.right != null) || (leaf && (h.left != null || h.right != null))) {
                        return false;
                }
                if (h.left != null) {
                        queue[r++] = h.left;
                }
                if (h.right != null) {
                        queue[r++] = h.right;
                }
                if (h.left == null || h.right == null) {
                        leaf = true;
                }
        }
        return true;
}
```

Binary Search Tree

1. Key value of left subtree is smaller, right subtree is larger.
2. Operations: searchKey, findMin, findMax, predecessor, successor, insert, delete
3. Running time of basic operations:
   On average: $\Theta(\log n)$, the expected height of the tree is logn.
   In the worst case: $\Theta(n)$, the tree is a linear chain of n nodes
4. Successor: find the node with the smallest key greater than key[x].
   Case 1: right(x) is non empty, successor(x) = the minimum in right(x)
   Case 2: right(x) is empty, go up the tree until the current node is a left child: successor(x) is the parent of the current node. If cannot go further, x is the largest element.
5. Predecessor: find the node with the largest key smaller than key[x].
   Case 1: left(x) is non empty, predecessor(x) = the maximum in left(x)
   Case 2: left(x) is empty, go up the tree until the current node is a right child: predecessor(x) is the parent of the current node. If cannot go further, x is the smallest element.
6. Deletion: If the deleted z has two children, find z's successor y (leftmost node in z's right subtree), y has either no or one right child (but no left child), delete y from the tree, replace z's key by y's key, and satellite data with y's.

Operation	BST	Sorted-array-based List	Linked List
Constructor	O(1)	O(1)	O(1)
IsFull	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)
RetrieveItem	O(logN)*	O(logN)	O(N)
InsertItem	O(logN)*	O(N)	O(N)
DeleteItem	O(logN)*	O(N)	O(N)

7. Find LCA on a common binary tree.

```
public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) {
                // 遇到空，或者p，或者q，直接返回
                return root;
        }
        TreeNode l = lowestCommonAncestor(root.left, p, q);
        TreeNode r = lowestCommonAncestor(root.right, p, q);
        if (l != null && r != null) {
                // 左树也搜到，右树也搜到，返回root
                return root;
        }
        if (l == null && r == null) {
                // 都没搜到返回空
                return null;
        }
        // l和r一个为空，一个不为空
        // 返回不空的那个
        return l != null ? l : r;
}
```

8. Find LCA on a binary search tree.

```
public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // root从上到下
        // 如果先遇到了p，说明p是答案
        // 如果先遇到了q，说明q是答案
        // 如果root在p~q的值之间，不用管p和q谁大谁小，只要root在中间，那么此时的root就是答案
        // 如果root在p~q的值的左侧，那么root往右移动
        // 如果root在p~q的值的右侧，那么root往左移动
        while (root.val != p.val && root.val != q.val) {
                if (Math.min(p.val, q.val) < root.val && root.val < Math.max(p.val, q.val)) {
                        break;
                }
                root = root.val < Math.min(p.val, q.val) ? root.right : root.left;
        }
        return root;
}
```

9. How to verify the binary search tree (BST)? Use inorder traverse!

```
public static int MAXN = 10001;
public static TreeNode[] stack = new TreeNode[MAXN];
public static int r;

public static boolean isValidBST(TreeNode head) {
    if (head == null) {
        return true;
    }
    TreeNode pre = null;
    r = 0;
    while (r > 0 || head != null) {
        if (head != null) {
            stack[r++] = head;
            head = head.left;
        } else {
            head = stack[--r];
            if (pre != null && pre.val >= head.val) {
                return false;
            }
            pre = head;
            head = head.right;
        }
    }
    return true;
}
```