Chun-Wei Chen

Assignment #7

03/15/12

1. (1) f(n) = O(n)

(2) f(n) = O(2n)

(3) f(n) = O(n2)

(4) f(n) = O(n·log(n))

1. Part 1 & Part 2 On the last page

Part 3:

**public void heapInsert(int o) {**

**ensureCapacity(size + 1);**

**int idx = size;**

**items[idx] = o;**

**while (idx > 0) {**

**if (o > items[(idx - 1) / 2]) {**

**int temp = items[(idx - 1) / 2];**

**items[(idx - 1) / 2] = items[idx];**

**items[idx] = temp;**

**idx = (idx - 1) / 2;**

**} else {**

**break;**

**}**

**}**

**}**

1. ArrayList: Create a queue: construct a new array (O(1)).

add(item): If the ArrayList is not full, directly add at the end of the list (O(n)); if the ArrayList is full, we need to create a new array with a larger capacity (usually double the current capacity) for the ArrayList in order to store more items, copy all the items of the current array into new array, and then add the new item (O(n));

get(0): Use index of the array to get the first item (O(1)).

remove(0): This operation needs to shift all the rest of the items in the ArrayList by one index (O(n)).

LinkedList: Create a queue: construct a new LinkedList (O(1)).

add(item): Use the pointer which points to the last item in the LinkedList to add the new item at the end (ex: last.next = Node(item);) (O(1)).

get(0): Use the pointer which points to the first item in the LinkedList to get the item directly (O(1)).

remove(0): Make the pointer which points to the first item in the LinkedList point to the second item (ex: first = first.next;) (O(1)).

Create a queue and get(0) are equally efficient for ArrayList and LinkedList. add(item) for LinkedList is sometimes more efficient than ArrayList. remove(0) for LinkedList is always more efficient than ArrayList. Therefore, using LinkedList to implement Queue is more efficient.

1. Stack: Last in first out

Create a stack: List<E> myStack = new ArrayList<E>()/LinkedList<E>();

push(item): Add the new item at the end -> add(item)

peek(): Get the item at the top of the stack without changing the stack

-> get(myStack .size() - 1) for ArrayList and get(0) for LinkedList

pop(): Remove the item at the top of the stack and returns that item’s value

-> remove(myStack .size() – 1) for ArraList and remove(0) for LinkedList

I choose the end of the ArrayList and the front of the LinkedList to be the top of the stack because it’s more efficient.

Let me use ArrayList to demonstrate.

List<Integer> myStack = new ArrayList<Integer>();

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

myStack.add(5); // push(5)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5(top) |  |  |  |  |

myStack.add(10); // push(10)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 10(top) |  |  |  |

myStack.add(32); // push(32)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 10 | 32(top) |  |  |

myStack.get(myStack.size() – 1); // peek() (Size is 3 right now. This doesn’t change the stack.)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 10 | 32(top) |  |  |

myStack.remove(myStack.size() – 1); // pop() (This changes the stack.)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 10(top) |  |  |  |

public void TreeBreadthFirst(Node p) {

if (p == null) {

List<Node> q = new LinkedList<Node>();

System.out.print(p.data);

q.add(p);

Node temp;

while(q.get(0) != null) {

temp = q.remove(0);

if (temp.left != null) {

System.out.print("," + temp.left.data);

q.add(temp.left);

}

if (temp.right != null) {

System.out.print("," + temp.right.data);

q.add(temp.right);

}

}

System.out.println();

}

}

The worst case scenario: perfect balanced binary tree. Every loop goes through each node and its two children (3n); therefore, the time complexity for this method is O(n).

6.

public static boolean isBalanced(String s) {

if (s != null && s.length() != 0) {

if (s.length() == 1) {

return !(s.equals("{") || s.equals("}") || s.equals("[") ||

s.equals("]") || s.equals("(") || s.equals(")"));

}

List<Character> stack= new ArrayList<Character>();

char c;

for (int i = 0; i < s.length(); i++) {

c = s.charAt(i);

if (c == '{' || c == '[' || c == '(') {

if (s.length() - i <= 2)

return false;

else if ((s.charAt(i) == '{' && s.charAt(i + 1) == '}') ||

(s.charAt(i) == '[' && s.charAt(i + 1) == ']') ||

(s.charAt(i) == '(' && s.charAt(i + 1) == ')'))

return false;

else

stack.add(c);

} else if (c == '}' || c == ']' || c == ')') {

if (i < 2 || stack.size() == 0)

return false;

else if ((s.charAt(i) == '}' && stack.get(stack.size() - 1) != '{') ||

(s.charAt(i) == ']' && stack.get(stack.size() - 1) != '[') ||

(s.charAt(i) == ')' && stack.get(stack.size() - 1) != '('))

return false;

else

stack.remove(stack.size() - 1);

}

}

if (stack.size() != 0)

return false;

}

return true;

}