**Advanced Algorithms**

**Exercise for Lecture 8,9,10**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name** |  | **Student ID** |  |
| **Lecture 5** |  | | |
| **Lecture 6** |  | | |
| **Lecture 7** |  | | |
| **Total Score** |  | | |
| **Notes** | Deadline: **2024-10-17 24:00**  Submission Format: ‘**Lecture567\_Name\_ID.docx**’, and please send to: **3459996503@qq.com**.  This assignment is meant to be an evaluation of your **individual** understanding coming into the course and should be completed **without collaboration** or outside help. | | |

**Lecture 8**

**Problem 8.1[15 points]** Observe that whenever we reference the size attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose we store in each node its rank in the subtree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

**Solution**:

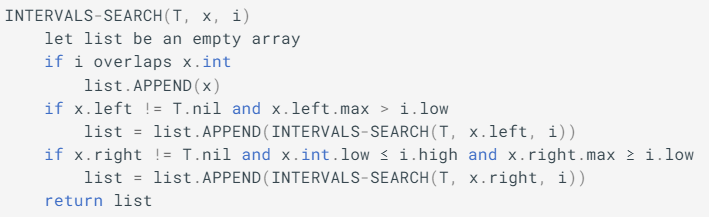
First perform the usual BST insertion procedure on z, the node to be inserted. Then add 1 to the rank of every node on the path from the root to z such that z is in the left subtree of that node. Since the added node is a leaf, it will have no subtrees so its rank will always be 1.

When a left rotation is performed on x, its rank within its subtree will remain the same. The rank of x.right will be increased by the rank of x. If we perform a right rotation on a node y, its rank will decrement by y.left.rank. The rank of y.left will remain unchanged.

For deletion of z, decrement the rank of every node on the path from z to the root such that z is in the left subtree of that node. For any rotations, use the same rules as before.

**Problem 8.2[10 points]** Given an interval tree T and an interval i, describe how to list all intervals in T that overlap i in O(min(n,klgn)) time, where k is the number of intervals in the output list. (Hint: One simple method makes several queries, modifying the tree between queries. A slightly more complicated method does not modify the tree.)

**Solution**:

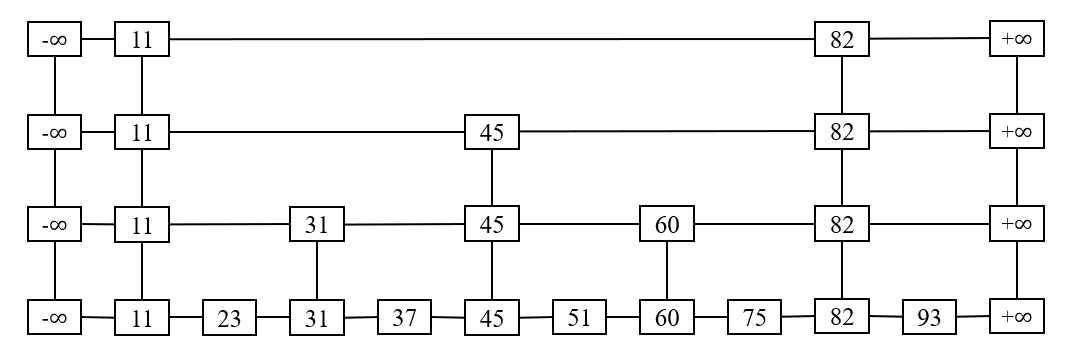


**Lecture 9**

**Problem 9.2[20 points]**

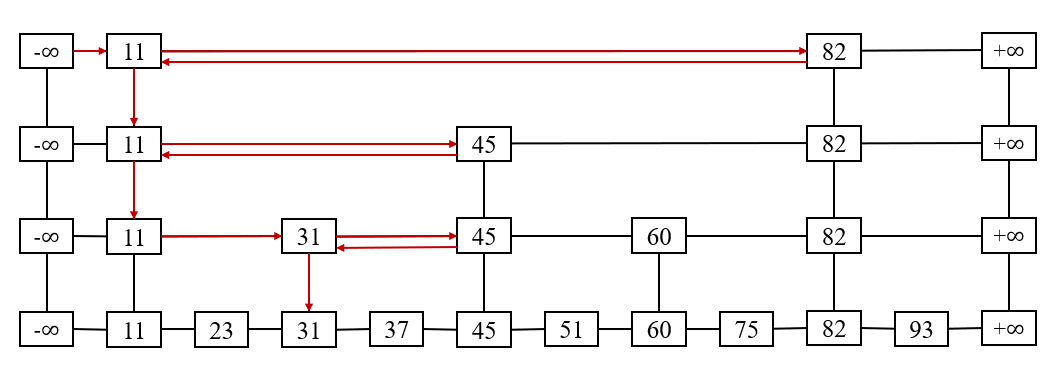
(1) Please show how to search 51 in the next skip list. Each comparison is an intermediate step, and you need to provide each step. The answer can be shown in one picture and the example is shown as follows:

(2) Please show how to search 93 in the next skip list.



**Picture 1. The skip list**

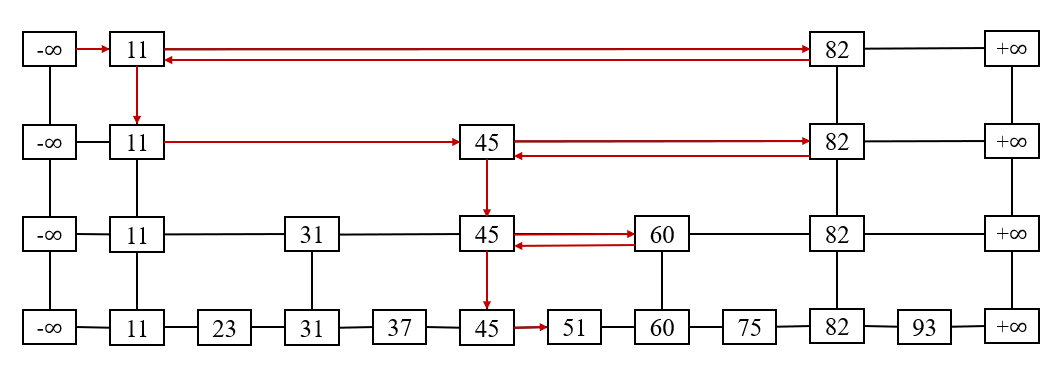
**Search 31**



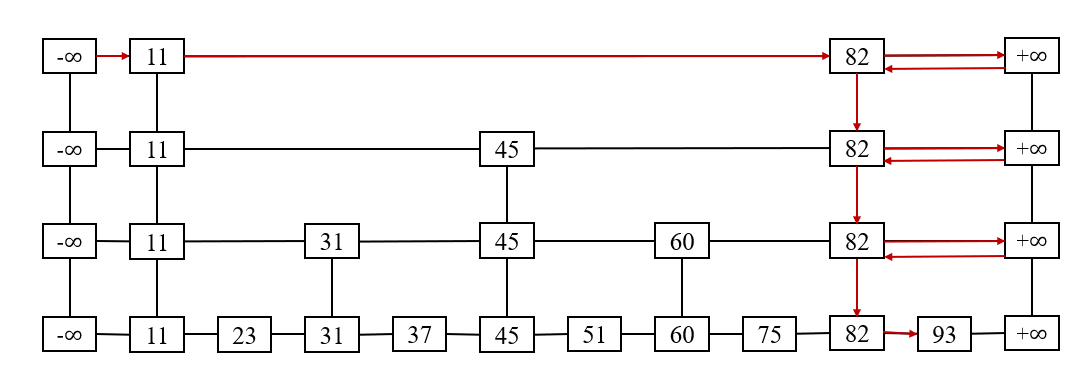
**Picture 2. The example of solution**

**Solution:**

**Search 51**



**Search 93**



**Problem 9.2[20 points]**

(1) Please design the data structure of the nodes in the skip list and the data structure of skip list separately.

(2) Please give the code for inserting operation in the skip list. Please give the effective C code directly instead of pseudo code whose name should be Skip\_List\_Delete().

**Solutions**:  
(1)

The data structure of nodes in the skip list.

typedef struct node

{

int key;

struct node \*next[1];

} Node;

The data structure of the skip list.

typedef struct skiplist

{

int level;

Node \*head;

} Skiplist;

(2)

bool Skip\_List\_Delete(Skiplist \*sl, int key) {

Node \*update[MAX\_LEVEL]; // keep track of updated nodes at each level

Node \*p = sl->head;

// Find the node and record the path

for (int i = sl->level - 1; i >= 0; i--) {

while (p->next[i] && p->next[i]->key < key) {

p = p->next[i];

}

update[i] = p; // Record the current node at this level

}

// Check if the target node exists

p = p->next[0];

if (p && p->key == key) {

// If it exists, delete the node

for (int i = 0; i < sl->level; i++) {

if (update[i]->next[i] != p) {

break; // If the node is not present at the current level, exit

}

update[i]->next[i] = p->next[i]; // Update pointers

}

// Free the memory of the node

free(p);

// If the top level's node is deleted, update the skip list level

while (sl->level > 0 && sl->head->next[sl->level - 1] == NULL) {

sl->level--;

}

return true; // Successfully deleted

}

return false; // Node does not exist

}

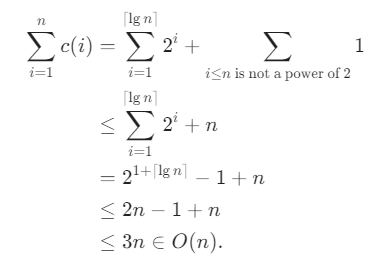
**Lecture 10**

**Problem 10.1[20 points]** Suppose we perform a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2,and 1 otherwise. Choose any two of three methods to determine the amortized cost per operation, respectively.

**Solution**:

• Aggregate Method

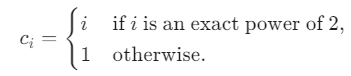
Let n be arbitrary, and have the cost of operation i be c(i). Then we have,



To find the average, we divide by n, and the amortized cost per operation is O(1).

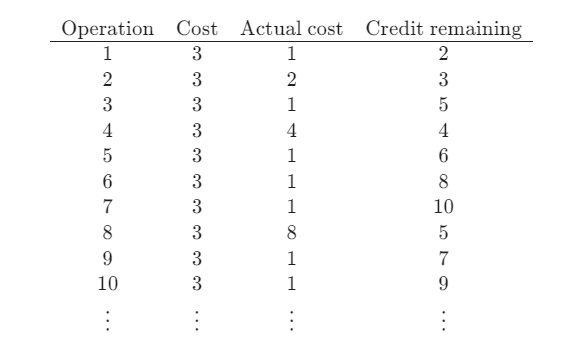
• Accounting Method

Let ci​= cost of ith operation.



Charge 3 (amortized cost ​) for each operation.

* If i is not an exact power of 2, pay $1, and store $2 as credit.
* If i is an exact power of 2, pay $i, using stored credit.



Since the amortized cost is $3 per operation, .

We know from aggragate method that .

Then we have

.

Since the amortized cost of each operation is O(1), and the amount of credit never goes negative, the total cost of n operations is O(n).

• Potential Method

Define the potential function Φ(*D*0​)=0, and Φ(*Di*​)=2*i*−21+⌊lg*i*⌋ for *i*>0.

For operation 1,

.

For operation *i*(*i*>1), if *i* is not a power of 2, then

If  for some  *j*∈N, then

.

Thus, the amortized cost is 3 per operation.

**Problem 10.2[15 points]** Show how to implement a queue with two ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUEDEQUEUE operation is O(1).

**Solution**:

Name the two STACKs as Stack1 and Stack2, we can implement the QUEUE as follows:

• ENQUEUE(x): PUSH x into Stack1

• DEQUEUE(x): If Stack2 is not empty, then simply POP from Stack2 and return the element. If Stack2 is empty, POP all the elements of Stack1, PUSH them into Stack2, then POP from Stack2 and return the result.

• Aggregate method

Consider a sequence of n operations. The sequence of operations will involve at most n elements. The cost associated with each element will be at most 4 i.e. (pushed into Stack1, popped from Stack1, pushed to Stack2, and popped from Stack2). Hence, the actual cost of n operations will be upper bounded by T(n)= 4 n. Hence, the amortized cost of each operation can be T(n)/n = 4n / n = 4 = O(1).

• Accounting method

We guess that the amortized costs for ENQUEUE and DEQUEUE are 3 and 1. We show that the potential function P(n) satisfies P(n) - P(0) >= 0 for all n. We have P(0) = 0. If an element is not popped, then it's only pushed twice and popped once. Thus, the cost of 3 is paid for by ENQUEUE operation. The cost for last pop operation is paid for by the DEQUEUE.

Note: Alternatively, we can set the costs for ENQUEUE and DEQUEUE as 4 and 0 respectively.

• Potential method

We guess the potential function P(n) = 2 \* #Elements in Stack1. P(0) = 0 and P(n) - P(0) >= 0 for all n.

ENQUEUE: Actual cost of PUSH is 1. Number of elements in Stack1 increases by 1 and Delta P increases by 2. Amortized cost = actual cost + ΔP = 1 + 2 = 3.

DEQUEUE:

If Stack2 is not empty. Actual cost of DEQUEUE is 1. The #Element in Stack1 stays the same, i.e. ΔP = 0. Amortized cost = actual cost + ΔP = 1 + 0 = 1.

If Stack2 is empty. Let x = #Elements in Stack1. The actual cost of POP is 2x. The ΔP = 0 - 2x = - 2x. Amortized cost = actual cost + ΔP = (2x+1) + (-2x) = 1.

Therefore, the amortized costs for ENQUEUE and DEQUEUE are 3 and 1 respectively.