Class Meeting #9

COS 226 — Spring 2018

Mark Braverman

Time and location

The midterm is during lecture on

Monday, March 12, 11–12:20pm.

The exam will start and end promptly, so please do arrive on time.

The midterm rooms are

Precepts P01, P02, P02A, P05, P05A: McDonnell 02 (new room)

Precepts P03, P04, P04A, P04B, P05B : Friend 101 (traditional lecture room)

Space is tight. Failure to go to the right room can result in a 10% deduction on the exam.

There will be no makeup exams except under extraordinary circumstances, which must be accompanied by the recommendation of a Dean.

Rules

- Closed book, closed note.
- You may bring one 8.5-by-11 sheet (one side) with notes in your own handwriting to the exam.
- No electronic devices (including calculators, laptops, and cell phones). Headphones attached to audio devices are also prohibited.
- Discussing the exam with others until solutions are posted is a violation of the Honor Code.

General considerations

Three types of questions:

- Quizzera style questions.
- Short questions, such as “what data structure to use”, “debug this code”.
- Questions involving design/implementation.
- Programming assignments are part of the material.

A useful review deck from last semester:

http://www.cs.princeton.edu/courses/archive/fall17/cos226/exams/midterm_review_f17.pdf

Design

- If performance guarantees are required, pay attention to them, often they hint at the implementation.
- Pay attention to any partial implementation given.
- Sometimes more than one data structure needs to be combined.
- Caching is another very useful technique (a special case of “more than one data structure”).

Data structures

Arrays	**Random access, easy to use, resizing needed**
Singly LL	**Flexible memory management, linear access**
Doubly LL	**Flexible memory management, extra memory, two-way access**
Weighted UF	**Linear memory, logarithmic union/find**
Queue/stacks	**Specialized operations (push, pop, enqueue, dequeue), LL or array implementation**
BST	**Supports order operations, bad worst case**
LLRB	**Supports order operations, all log n operations**
max/min heap	**log n inserts/deletes, constant find min/max**
ST	**BST, LLRB, hashtable implementations (not yet covered)**

Warm-up

How to implement a max-PQ with an LLRB BST?

Same performance as heap implementation:

- Insert/delete in time proportional to log n
- Constant max()

Solution

Cache the maximum value.

Design question

Randomized priority queue. (8 points)

Describe how to add the methods `sample()` and `delRandom()` to our binary heap implementation of the `MinPQ` API. The two methods return a key that is chosen uniformly at random among the remaining keys, with the latter method also removing that key.

public class MinPQ<Key extends Comparable<Key>>		
	`MinPQ()`	*create an empty priority queue*
`void`	`insert(Key key)`	*insert a key into the priority queue*
`Key`	`min()`	*return the smallest key*
`Key`	`delMin()`	*return and remove the smallest key*
`Key`	`sample()`	*return a key that is chosen uniformly at random*
`Key`	`delRandom()`	*return and remove a key that is chosen uniformly at random*

Design question

You should implement the `sample()` method in constant time and the `delRandom()` method in time proportional to $\log N$, where N is the number of keys in the data structure. For simplicity, do not worry about resizing the underlying array.

Your answer will be graded on correctness, efficiency, clarity, and conciseness.

Design question

```
public Key sample() {
    int r = 1 + StdRandom.uniform(N);  // between 1 and N
    return a[r];
}
```

Design question

```
public Key delRandom() {
    int r = 1 + StdRandom.uniform(N);  // between 1 and N
    Key key = a[r];                    // save away
    exch(r, N--);                      // to make deleting easy
    sink(r);                           // if a[N] was too big
    swim(r);                           // if a[N] was too small
    a[N+1] = null;                     // avoid loitering
    return key;
}
```

Design question

We would like to have a *capacitated stack, with max capacity k.*
Supported operations

```
public class CapStack <Item>
public CapStack(int cap); // create stack of
                                    given capacity
public Item pop();
public void push(Item item);
```

- The stack only stores at most cap items.
- If more items are pushed, oldest items are discarded.
- All operations should be constant time.
- Memory should be proportional to current stack size.

Solution

Use a dequeue.
CapStack will have instance variables:

```
int capacity;
Deque<Item> data;
public CapStack(int cap)
{
 capacity = cap;
 data = new Dequeue();
}
```

```
public Item pop()
{
   return data.removeFirst();
}
```

```
public Item push(Item item)
{
   data.addFirst(item);
   if (data.size()>cap)
      data.removeLast();
}
```

Follow-up questions

- Dequeue can be implemented as a linked list, a doubling resizable array, or a fixed-size array.
- Which implementations are consistent with the memory requirement?
- Suppose that the stack is full to capacity most of the time.
- Order the implementations in terms of expected memory performance in practice.

Design question

Given k sorted arrays containing N keys in total, design an algorithm that determine whether there is any key that appears more than once.

Your algorithm should use extra space at most proportional to k. For full credit, it should run in time at most proportional to $N \log k$ in the worst case; for partial credit, time proportional to Nk.

Design question

The main idea is to consider the N keys in ascending order, so that duplicate keys are adjacent. This is similar to the multiway merging algorithm on pp. 321–322 of the textbook.

- Scan through adjacent entries in each of the k sorted array to check for any duplicate key within one of the original sorted arrays. If a duplicate is detected, stop.
- Initialize a red-black BST with k key-value pairs, where the key is the first (smallest) key in the ith sorted array and the value is the index i of the array.
- Repeat until the BST is empty or a duplicate is detected:
  - Delete the minimum key from the BST and let i be the index of the array associated with the deleted key.
  - If the next remaining key from array i is not already in the BST, add the key and associate it with the value i.
  - Otherwise, stop (duplicate detected).

Other solutions?

Design question

8. Addendum (9 points). The `addBlock()` operation is used to add M `comparables` to an existing sorted data set of N `comparables`, where M << N. A data set of size N is considered sorted if it can be iterated through in sorted order in N time.

COS226 student Frankie Halfbean makes two choices. First, he selects a sorted array as the data structure. Secondly, he selects insertion sort as the core algorithm, explaining that insertion sort is very fast for almost sorted arrays. To add a new block of M comparables, the algorithm simply creates an array of length N+M, copies over the old N values into the new array, copies over the new M values to the end of the array, and finally insertion sort is used to bring everything into order. The old array is left available for garbage collection.

(a) What is the worst case order of growth of the run time as a function of N and M?

Design question

(b) Design a scheme that has a better order of growth for the run time in the worst case. For full credit, design a scheme that uses optimal space and time to within a constant factor.

Design question

Largest common item. (8 points)

Given an N-by-N matrix of real numbers, find the largest number that appears (at least) once in each row (or report that no such number exists).

9	6	3	8	5
3	5	1	6	8
0	7	5	3	5
3	5	7	8	6
4	3	5	7	9

The running time of your algorithm should be proportional to $N^2 \log N$ *in the worst case. You may use extra space proportional to* N^2.

(a)
1. Sort each row using heapsort.
2. For each number in row 0, from largest to smallest, use binary search to check if it appears in the other $N - 1$ rows.
3. Return the first number that appears in all N rows.

The order of growth of the running time is $N^2 \log N$, with the bottleneck being steps 1 and 2. Correctness follows because the largest common number must appear in row 0. Scanning the numbers in row 0 from largest to smallest ensures that we find the *largest* common number.

(b) $N^2 \log N$

7. **Leaky stack. (8 points)**

A *leaky stack* is a generalization of a stack that supports adding a string; removing the most-recently added string; and deleting a random string, as in the following API:

public class LeakyStack	
LeakyStack()	*create an empty randomized stack*
void push(String item)	*push the string on the randomized stack*
String pop()	*remove and return the string most recently added*
void leak()	*remove a string from the stack, uniformly at random*

All operations should take time proportional to $\log N$ *in the worst case, where* N *is the number of items in the data structure.*

```
public class LeakyStack {
    private int counter = 0;
    private RedBlackBST<Integer, String> st = new RedBlackBST<Integer, String>();

    public void push(String item) {
        st.put(counter++, item);
    }

    public String pop() {
        String item = st.get(st.max());
        st.deleteMax();
        return item;
    }

    public void leak() {
        int r = StdRandom.uniform(st.size());
        st.delete(st.select(r));
    }
}
```

You have been hired by Deep Thought Enterprises to implement a priority-queue-like data structure supporting the following operations:

- `insert()` an item in $O(\log N)$ time.
- `fortytwo()` — return the 42^{nd} smallest item in constant time.
- `delFortyTwo()` — delete the 42^{nd} smallest item in $O(\log N)$ time.

Explain how you would implement the required functionality, using one or more data structures that we have seen in class. Write pseudocode for each of the three operations listed above. You may assume that $N > 42$, and omit all checks for smaller N.

For full credit, your implementation should support finding the k^{th} smallest item with an order-of-growth running time ***independent*** of k. That is, it should be possible to change 42 to some other constant (at compile time) without changing the order-of-growth running time.

Solution #1:
Maintain a MaxPQ called firstFortyTwo with 42 items on it,
and a MinPQ called theRest with all the rest of the items.

```
insert(x):
        firstFortyTwo.insert(x);
        if (firstFortyTwo.size() > 42)
            theRest.insert(firstFortyTwo.delMax());
fortytwo():
        if (firstFortyTwo.size() < 42) // Check may be omitted in answer
            throw new NoSuchElementException("N < 42");
        return firstFortyTwo.max();
delFortyTwo():
        if (firstFortyTwo.size() < 42) // Check may be omitted in answer
            throw new NoSuchElementException("N < 42");
        x = firstFortyTwo.delMax();
        if (!theRest.isEmpty())
            firstFortyTwo.insert(theRest.delMin());
        return x;
```

Solution #2:
Maintain a Red-Black BST, as well as the 42nd-smallest element.
insert() adds to the RBST, then calls select(42) to find the
    42nd-smallest element, saving it in an instance variable.
fortytwo() returns the cached 42nd-smallest element.
delFortyTwo() deletes the 42nd-smallest element from the RBST, then calls
    select(42) to save the new 42nd-smallest element.

Incorrect solution:
Maintain a Red-Black BST, but call select(42) in fortytwo() - takes log(N) time.