Lecture 4

Data Structures Review

1. What are the main operations supported by a red-black tree, what are their runtimes, and what are the Java/C++ classes that implement this functionality?

2. Can you find the kth largest element of a red-black tree quickly? If yes, explain how, and which Java/C++ function you would call.

3. What primary operations does a priority queue support, what are their runtimes, how do you access them in each language, and how is it implemented?

4. How do you sort a static array in Java/C++? What algorithm is used? What is a stable sort, and how do you do it in Java/C++?

5. How do you use a hashtable in Java/C++? What is the performance?

6. How do you use a dynamically sized array in C++/Java? What is the runtime performance of the standard operations?

7. What are the operations that the union-find data structure supports and what is the runtime? What is the cost of performing n merge/find operations on the union-find data structure?

Solutions

1. Test membership, insert, and remove are all possible in $O(\lg n)$ time. The Java generic classes TreeSet,TreeMap and the C++ template classes map,set implement the functionality.

2. No. If we augment the red-black tree to store the size of each subtree in each node, then yes. There is no Java/C++ function.

3. A priority queue supports insert and pop, both typically having runtime $O(\lg n)$. They are implemented as a binary heap (a "nearly" complete binary tree stored in contiguous memory, and operated on using sift up/down; can be constructed in linear time). In Java you can use PriorityQueue. In C++ you can use make_heap, push_heap, pop_heap in the algorithm header.

4. In Java, Arrays.sort, in C++, sort in the algorithm header. In Java, a variant of quicksort is used that has expected runtime $O(n \lg n)$ and worst-case runtime $O(n^2)$. The C++ algorithm typically used is a mixture of quicksort, insertion sort, and heapsort

(called introsort) giving worst case runtime $O(n \lg n)$ (in C++11 the worst-case runtime is guaranteed). A stable sort is a sort that leaves values with the same sort key ordered by their position in the original list. To stable sort in Java you must use Collections.sort on a list. To stable sort in C++ you use stable_sort in the algorithm header. Both typically use a variant of merge sort.

5. In Java you use HashMap, or HashSet. In C++ you don't have access to a hashed container (before C++11; after, use unordered_map/unordered_set).

6. In Java you use ArrayList, in C++ you use vector. Removal from the end, and element access is always $O(1)$. Other removals, and insertion are always worst-case runtime $O(n)$. Insertion at the end has amortized runtime $O(1)$.

7. Given two elements, union the sets they belong to. Given an element, find the canonical representative for the set it belongs to. Union is $O(1)$ and find is worst case $O(\lg n)$ (implemented as disjoint forest with height heuristic). The cost of n merge/find operations is $O(n\alpha(n))$ where α is the inverse Ackermann function (effectively $O(n)$; requires path compression).

Bitmasks

Consider the set of integers $S = \{0, 1, \ldots, n\}$. We want an efficient data structure that will represent a subset of S and supports $O(1)$ operations like membership testing, insertion, removal, intersection, union, and set subtraction. To do this we will restrict $n < 64$ and store our set in a 64-bit integer type (for larger sets, use the language supported classes). Each element of S corresponds to a bit of the integer. The lowest order bit is used for 0, the kth lowest bit is used for k. To implement the set operations we will use the bitwise operators xor (^), and (&), or (|), and not (~). Occasionally it is useful to make use of the two-complement representation by using the unary negation operator $(-)$, and addition $(+)$. Most of the functionality of bitsets is implemented by the C++ bitset and Java Bitset, although the C++ implementation is fixed size while the Java implementation grows as necessary (both can be arbitrarily large).

Exercises

1. Show how insertion, removal, membership testing, intersection, union, and set subtraction are implemented.

2. Show how to insert every element into the set.

3. Show how to remove the smallest member of a set.

4. Show how to get the value of the least significant one of an integer.

5. Show how to count the number of elements in the set.

6. What are the values of 2<<33, 2>>33, 1L<<33, 1<<(−30), (~0)>>1, (~0)>>>1 in Java? in C++?

7. Show how to remove the trailing contiguous sequence of ones from a number (if it exists)? That is, $10110111 \mapsto 10110000$ and $11000 \mapsto 11000$.

8. Show how to iterate through all subsets, where $n = 20$.

9. Fill in the blank with bit operations so that the following code iterates through all permutations of $\{0, 1, 2, 3\}$.

```
for (int a = 0; a < 4; ++a)
  for (int b = 0; b < 4; ++b)
    for (int c = 0; c < 4; ++c)
      for (int d = 0; d < 4; ++d)
      {
          if (________________)
          {
            //Process permutation a,b,c,d
          }
      }
```

Solutions

Let **long** s, t represent sets and let $0 \leq k < 64$ (in C++ use long long and 1LL).

1. (a) Insert k: s |= (1L<<k).
   (b) Remove k: s &= ~(1L<<k).
   (c) Membership test for k: (s & (1L<<k))= 0!.
   (d) Intersect s, t: s & t.
   (e) Union s, t: s | t.
   (f) Subtract t from s: s & (~t).

2. Assuming numbers $0, \ldots, n$: s = (1L<<(n+1))−1 if $n \neq 63$ and s = −1L if $n = 63$.

3. s &= s−1

4. s & (−s)

5. Iterative:

```
int cardinality(long s)
{
  int count = 0;
  while (s != 0)
```

```
    {
      count++;
      s &= s-1;
    }
    return count;
  }
```

Recursive:

```
int card(long s) { return s == 0 ? 0 : 1 + card(s & (s-1)); }
```

6. (a) 2<<33: In Java, 4, in C++, undefined.
   (b) 2>>33: In Java, 1, in C++, undefined.
   (c) 1L<<33: In Java, the 64-bit value 2^{33}, in C++, a 32-bit undefined value (1LL or 1ULL is needed).
   (d) 1<<(−30): In Java, 4, in C++, undefined.
   (e) (~0)>>1: In Java, all 32 bits set, in C++, same.
   (f) (~0)>>>1: In Java, lowest 31 bits set, in C++, compile error.
7. s &= s+1
8.
```
for (int s = 0; s < (1<<20); ++s)
{
  //do work here
}
```
9. (1<<a)|(1<<b)|(1<<c)|(1<<d)== 0xF

Segment Trees

Suppose you have a list L of elements whose length is fixed, but whose entries may change. We are interested in querying statistics about continguous sublists of L. For example, we could want the min, or max of a range of elements. We could return the min or max value corresponding to the range, but it is even more informative if we return the index of an element that achieves that min or max value. We want a data structure that would let us make a particular type of range query (min or max) in $O(\lg n)$ time, and would let us update the list in $O(\lg n)$ time. To do this we build a segment tree. A segment tree is a nearly complete binary tree whose shape only depends on the number of elements in the list (similar to the binary heap). Each node of the binary tree represents a closed interval $[a, b]$ of indices. Leaves represent intervals $[a, a]$ containing one element. Each internal node corresponding to the range $[a, b]$ has two children. The left child has range $[a, \lfloor (a + b)/2 \rfloor]$ and the right child has range $[\lfloor (a + b)/2 \rfloor + 1, b]$. The value of each node is the result of the range query for the range of indices corresponding to that node. The height of the tree

is logarithmic in the size of the list. To make a range query, we traverse the tree using the following pseudocode (assuming indices are 0-based; tree is stored compactly in **int**[] st):

```
//Index of left child
int left(int n) { return 2*n+1; }
//Index of right child
int right(int n) { return left(n)+1; }

public int minQuery(int L, int R) { return
   minQuery(L,R,0,list.size()-1,0); }

//Get index of minimum value in index range [L,R]
//Current node n has index range [nL,nR]
public int minQuery(int L, int R, int nL, int nR, int n)
{
    if (L <= nL && nR <= R) return st[n];
    int lMin = -1, rMin = -1;
    int mid = (nL+nR)/2;
    if (L <= mid) lMin = minQuery(L,R,nL,mid,left(n));
    if (mid+1 <= R) rMin = minQuery(L,R,mid+1,nR,right(n));
    if (lMin == -1 || rMin == -1) return lMin == -1 ? rMin : lMin;
    return list.get(lMin) <= list.get(rMin) ? lMin : rMin;
}
```

It isn't hard to prove that this query will fully traverse at most two paths from root to leaf (split can only occur at the least common ancestor of the leaves corresponding to a, b). That said, at each node along these paths we may inspect both children. Thus the runtime is $O(\lg n)$. Updating the segment tree is as follows (again 0-based indices):

```
public void update(int pos, int value) {
   update(pos,value,0,list.size()-1,0); }

//Update segment tree at given position with given value.
//Current node n has index range [nL,nR]
public void update(int pos, int value, int nL, int nR, int n)
{
    if (nL == nR)
    {
        list.set(pos,value);
        st[n] = pos;
    }
    else
    {
        int mid = (nL + nR)/2, l = left(n), r = right(n);
        if (pos <= mid) update(pos,value,nL,mid,l);
        else update(pos,value,mid+1,nR,r);
        st[n] = list.get(st[l]) <= list.get(st[r]) ? st[l] : st[r];
```

```
        }
    }
```

As can be seen above, updating traverses a single root-to-leaf path, and thus has runtime $O(\lg n)$. Just like a binary heap, we can store a segment tree very compactly in an array where the ranges and child pointers are implicit. Also like a binary heap, we can construct a segment tree in linear time.

Fenwick (or Binary Indexed) Trees

If all you care about is sum range queries, there is a simplified tree which has a very short and efficient implementation. As before, we assume the total size of the list is constant, but the entries may change. The Fenwick tree (really just a table) will allow us to determine the sum of the numbers with indices in the range $[1, k]$ in $O(\lg n)$ time (we do everything 1-based in a Fenwick tree to make the code cleaner). An arbitrary range can then be achieved by subtracting the results from two ranges. Here we will just illustrate the idea behind the Fenwick tree, and how to implement it. Suppose k has the binary representation $11010_2 = 26_{10}$, i.e., we want the sum of the elements with indices in the range $[1, 26]$. To do this we could sum over the ranges:

$$[11001_2, 11010_2], \quad [10001_2, 11000_2], \quad [1_2, 10000_2].$$

Looking at the numbers 11010_2, 11000_2, and 10000_2 we see that the second and third arise from their predecessors by removing the least significant one bit. We stopped when there were no more bits to remove. More formally, suppose $k = 2^{e_n} + 2^{e_{n-1}} + \cdots + 2^{e_1}$ where $e_1 < e_2 < \cdots < e_n$. Let $\sum_{i=a}^{b} L[i]$ denote the sum of the list elements with indices in $[a, b]$. Then we have the equation:

$$\sum_{i=1}^{k} L[i] = \sum_{i=1}^{k-2^{e_1}} L[i] + \sum_{i=k-2^{e_1}+1}^{k} L[i].$$

We will store the rightmost sum in our table at index k. The leftmost sum is a range query on the "simpler" interval $[1, k - 2^{e_1}]$, which we can evaluate recursively. Here simpler means the upper limit of our interval has fewer one's in its binary representation. Let T denote the table representing our Fenwick tree. We will store the sum of the values with indices in $[k - 2^{e_1} + 1, k]$ at the location $T[k]$. Using the example from above, the three ranges will have sums stored as follows:

$$[11001_2, 11010_2] \to T[11010_2], \quad [10001_2, 11000_2] \to T[11000_2], \quad [1_2, 10000_2] \to T[10000_2].$$

From an earlier exercise, we already know how to remove the least significant one from an integer with a simple bitwise operation. This makes the query code extremely easy. To update the list at an index j we need to update all table entries corresponding to ranges that contain the index j. It can be shown that we can iterate through the necessary table entries by repeatedly adding the least significant one to our current index. The following pseudocode shows this (and has $O(\lg n)$ runtime):

```
void update(FenwickTable T, int index, int inc)
{
  for (;index < T.length; index += (index&(-index))) T[index] += inc;
}
```

Both of these operations will be fleshed out in the following exercises. Code can also be found in the book.

Exercises

1. Consider the (1-indexed) list $L = [1, -7, 3, 2, 1, -9]$.
   (a) Draw a min segment tree for L. Make sure to describe the interval and index value at each node.
   (b) Show how to query the min for (1-indexed) $[1, 1]$, $[4, 6]$ and $[2, 5]$.
   (c) Show what happens when $L[6]$ (1-indexed) is updated from -9 to 10.
2. How can you use a segment tree for sum range queries?
3. Consider the list $L = [1, 2, 1, 3, -1, 1]$.
   (a) Write the associated Fenwick tree table.
   (b) Find the sum of the range $[3, 6]$ using the table as the algorithm would.
   (c) Show what happens when $L[1]$ is updated from 1 to 4.
4. Give pseudocode for querying a Fenwick tree.
5. Suppose you have a Fenwick tree with 300 entries.
   (a) What table entries would you look at when computing the sum for the index range $[1, 107]$ (note $107_{10} = 1101011_2$)?
   (b) What table entries would you update when changing the value at entry 107?
6. Suppose you didn't want to store the actual list, and just the Fenwick tree's table. How could you retreive an element of the list? What is the runtime tradeoff?
7. Why can't we use a Fenwick tree for min and max queries?
8. Suppose you have a set of $n = 10^6$ of the numbers $\{1, 2, \ldots, n\}$. You are given a list of instructions each taking one of the following forms:
   (a) Remove the number k.
   (b) Output the jth largest remaining number.

   Explain how to use a Fenwick tree to deal with each instruction in $\log(n)$ time.

9. Suppose you are given a list of instructions regarding the multiset S each taking one of the following forms:

   (a) Insert the value k, where k is a 32-bit signed integer.

   (b) Remove the value k, where k is a 32-bit signed integer.

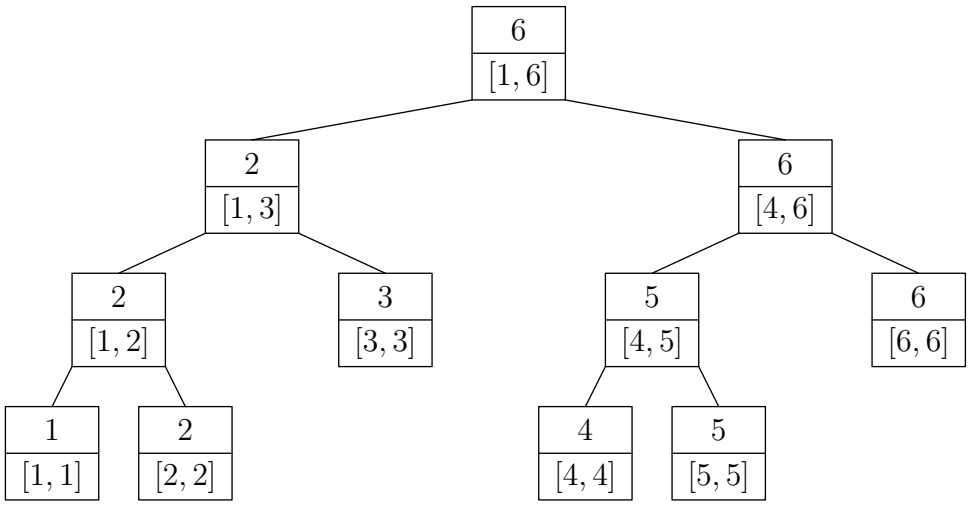
   (c) Give the percentage of the list smaller than x, where x is a 32-bit signed integer.

   (d) Give the jth largest element in the set.

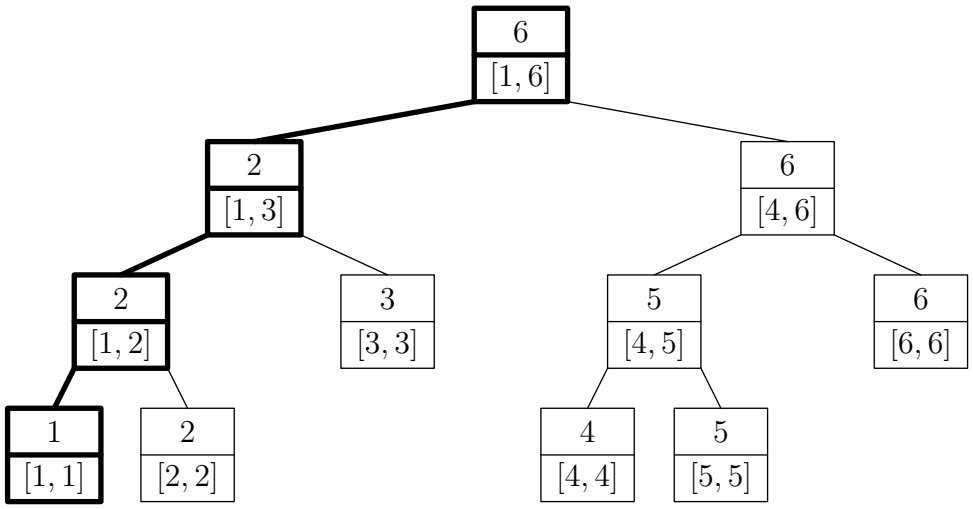
   Assume further that you are guaranteed only 10^6 distinct values will occur (but each may occur many times). Explain how to a Fenwick tree to efficiently handle these instructions.

Solutions

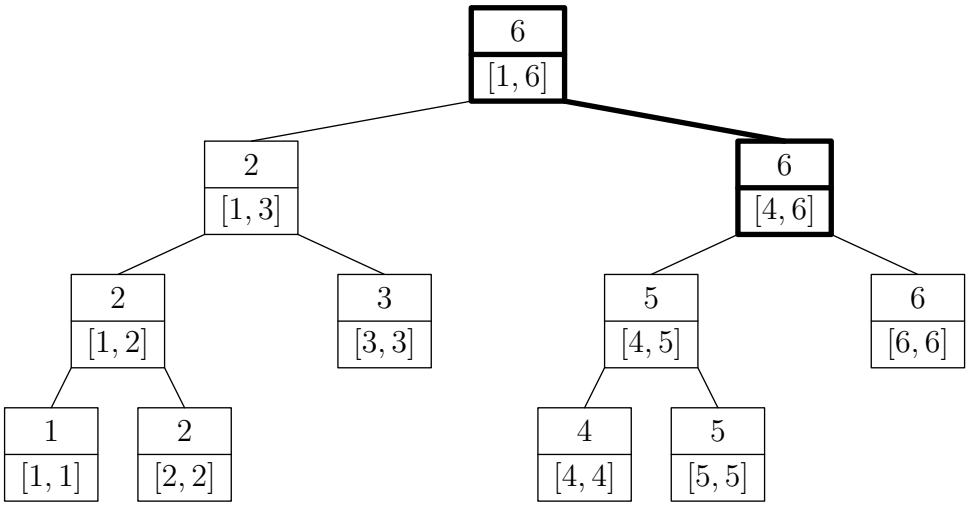
1. (a)



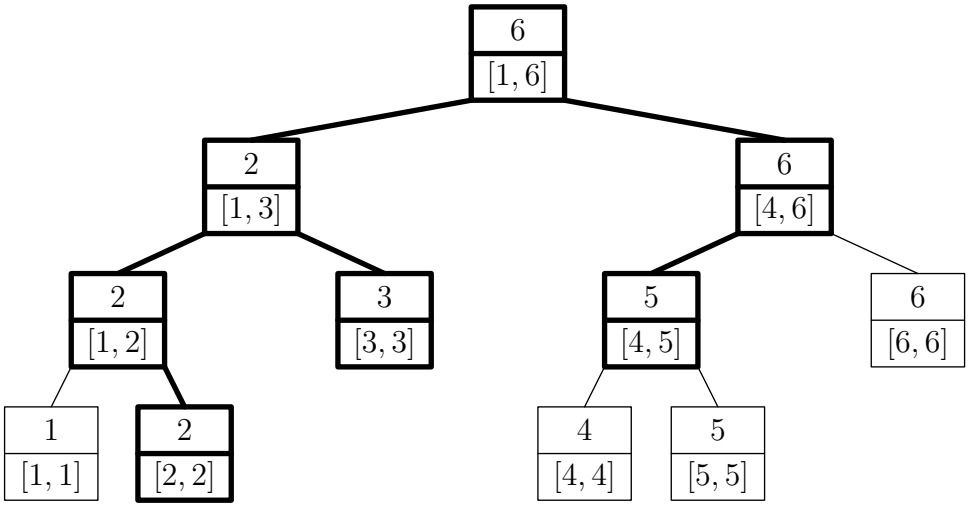
   (b) i. $[1, 1]$: Value 1 at index 1.



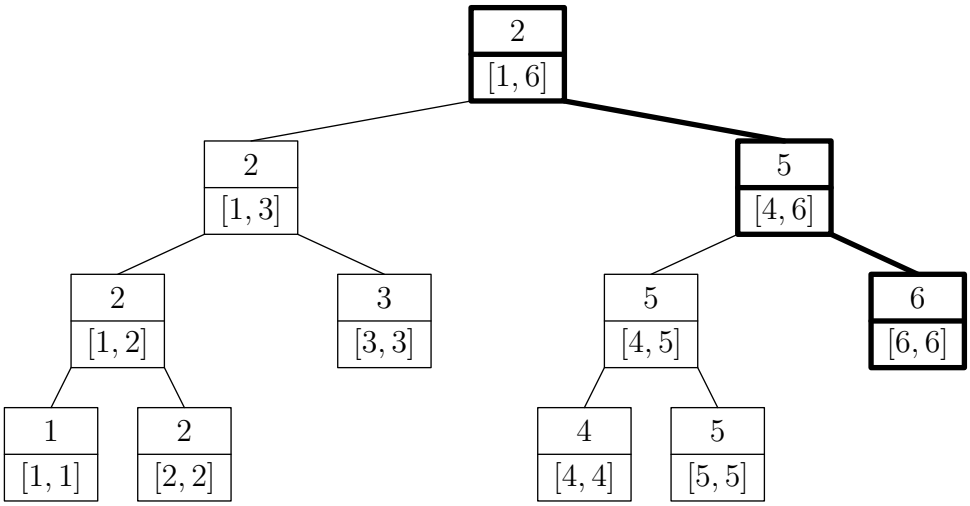
ii. $[4, 6]$: -9



iii. $[2, 5]$: -7



(c)



2. At each node, instead of storing the index of the minimum entry we store the sum.

3. (a) The table T is given below with extra information so that it is easier to understand.

Index_{10}	Index_2	Interval	Value
1	1	$[1, 1]$	1
2	10	$[1, 2]$	3
3	11	$[3, 3]$	1
4	100	$[1, 4]$	7
5	101	$[5, 5]$	-1
6	110	$[5, 6]$	0

   (b) We will compute the sum over $[1, 6]$ and subtract the sum over $[1, 2]$. The sum over $[1, 2]$ is $T[2] = 3$. The sum over $[1, 6]$ is $T[6] + T[4] = 7$. Note that removing the least significant one from 6 yields 4.

   (c) The resulting table follows with altered entries in bold.

Index_{10}	Index_2	Interval	Value
1	1	$[1, 1]$	**4**
2	10	$[1, 2]$	**6**
3	11	$[3, 3]$	1
4	100	$[1, 4]$	**10**
5	101	$[5, 5]$	-1
6	110	$[5, 6]$	0

4. Here is pseudocode to query the tree for the sum over $[1, k]$:

```
int query(FenwickTable T, int k)
{
  int ret = 0;
```

```
    while (k != 0)
    {
      ret += T[k];
      k &= k-1;
    }
    return ret;
  }
```

5. (a) For $107_{10} = 1101011_2$ we need to sum the entries with indices 1101011_2, 1101010_2, 1101000_2, 1100000_2, and 1000000_2 corresponding to the ranges

$$[107, 107], [105, 106], [97, 104], [65, 96], [1, 64].$$

   (b) When updating $107_{10} = 1101011_2$ we need to update indices 1101011_2, 1101100_2, 1110000_2, 10000000_2, 100000000_2.

6. By performing a sum range query on the interval $[k, k]$ you can get the value at index k. This has a $O(\lg n)$ runtime cost versus the $O(1)$ runtime cost of keeping the list around.

7. We could if we only wanted to use index ranges of the form $[1, k]$ (just store mins instead of sums). Otherwise, no. The minimum over the range $[a, b]$ cannot be computed quickly using min range queries of the form $[1, k]$.

8. Build a Fenwick tree for a list of size n where every element in the list L has the value 1. Here the entry $L[k]$ will denote whether or not k is currently in our set. Removals are performed by updating the list entry to 0. To find the jth largest remaining number, we must find the lowest index i such that the sum over the interval $[1, i]$ is j. This can be found using binary search by repeatedly making sum queries (possible since our list is entirely zeros and ones). The runtime is $O((\lg n)^2)$.

9. Building a Fenwick tree for every possible value is infeasible, so we will first compress the number range to have size 10^6. We accomplish this by preprocessing the list of instructions, collecting all of the distinct values in strictly increasing order. We then create a mapping from the integers $1, \ldots, 10^6$ to the distinct values preserving the increasing order (use an array for one direction, and a Tree/HashMap for the reverse). Construct a Fenwick tree for a list L of length 10^6.

   (a) To insert v we find the value $k \in [1, 10^6]$ that v corresponds with, and increment the list entry $L[k]$.

   (b) To remove we decrement the list in the same manner.

   (c) To compute the percentage of the list smaller than x, we binary search the list of distinct values to find the largest value v than is less than x, and then compute the value $k \in [1, 10^6]$ that it maps to. The percentage is computed by summing over $[1, k]$ and then dividing by the total number of elements in the set.

(d) To find the jth largest element, we binary search for the smallest value i such that the sum over $[1, i]$ is at least j. Then we map i to its actual value.