• **Homework 1**
  • Tomorrow: Friday, Feb 5
  • Due following Friday, Feb 12 at 2pm
  • Remember to register an account on Virtual Judge with a username of the form: 
    `netid_CS480S16`

• **Office Hours**
  • Wednesday mornings from 9am to 10am in WWH 328
  • Please email me in advance just to give me a heads up
• What are the main operations supported by a red-black tree?
• What are their runtimes?
• What are the Java/C++ classes that implement this?
• Can you find the $k$th largest element of a red-black tree quickly?
• What are the primary operations of a priority queue?
• What are their runtimes?
• What are the Java/C++ classes that implement this?
• How is this implemented?
• How do you sort a static array in Java/C++?
• What algorithm is used?
• A stable sort is a sort that leaves values with the same sort key ordered by their position in the original list
• Java: Collections.sort on a list
• C++: stable_sort in <algorithm>
• Both typically use a variant of merge sort.
• How do you use a hashtable in Java/C++? What is the performance?
• What is the cost of performing $n$ merge/find operations on the union-find data structure?
• Given a set $S$ of discrete elements, design an efficient data structure that will represent a subset of $S$ and:
  • Support operations like membership testing, insertion, removal, intersection, union, set subtraction all in $O(1)$ time
• For simplicity, $|S| \leq 64$
  • If $|S| > 64$, language-supported types: Bitset (Java) and <bitset> (C++)
  • <bitset> is fixed size, Bitset grows arbitrarily
• Use the bits of an integer, where each bit represents the membership of an element
• All set operations implemented using $\text{xor} \ ^{\land}$, and $\&$, or $|$ and not $\sim$
  • Sometimes addition $+$ and negation $-$
• Insertion: $s |= (1L << k)$
• Removal: $s &= ~(1L << k)$
• Membership testing: $s & (1L << k)$
• Intersection: $s \& t$
• Union: $s | t$
• Set subtraction: $s \& \neg t$
• How do you insert every element into a set of length s?
• How do you remove the smallest member of a set S?
• How do you get the value of the least significant one in a set?
• How do you count the number of elements in the set?
• What are the values of $2 \ll 33$, $2 \gg 33$, $1L \ll 33$, $1 \ll -30$, $(~0) \gg 1$, $(~0) \gg> 1$ in Java? In C++?
• How do you remove the trailing contiguous sequence of ones from a number (if it exists)
• How do you iterate through all subsets of a set S?
• Suppose you have a list $L$ of elements whose length is fixed, but whose entries may change
• Design a data structure that can efficiently return the min or max of a range of elements
  • Also return the index of this element
A segment tree is a nearly complete binary tree whose shape is determined by the number of elements.

Each node represents a closed interval $[a, b]$ of indices:

- Leaves represent intervals with a single element $[a, a]$.

Each internal node has two children:

- Left child: interval $[a, \text{floor}((a+b)/2)]$.
- Right child: interval $[\text{floor}((a+b)/2) + 1, b]$.

The value of each node corresponds to the range query for that interval.

Segment trees allow us to:

- Make a range query (min or max) in $O(\lg n)$ time.
- Update the list in $O(\lg n)$ time.
Draw a min segment tree for $[1, -7, 3, 2, 1, -9]$: (1-based indices)
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;
}
Query for intervals [1, 1] (1-based indices)
Query for intervals [4, 6] (1-based indices)
Query for intervals [2, 5] (1-based indices)
• Query will 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]
  • Exercise: how to efficiently find the least common ancestor of two nodes in a tree?
  • Each node along these paths may inspect both children
• Running time is \( O(\lg n) \)
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];
    }
}
Update L[6] from -9 to 10 (1-based index)
• Traverses a single root-to-leaf path
• Running time $O(\lg n)$
// Initialize: build(0, list.size() - 1, 0);
public void build(int nL, int nR, int n) {
    if (nL == nR)
        st[n] = nL;
    else {
        int mid = (nL + nR) / 2, l = left(n), r = right(n);
        build(nL, mid, l);
        build(mid + 1, nR, r);
        st[n] = list.get(st[l]) <= list.get(st[r]) ? st[l] : st[r];
    }
}
Competitive Programming 2.4.3