Selection and Hashing
The Selection Problem

• Input is a set of \( n \) distinct items \( \{a_1, a_2, \ldots, a_n\} \) and an integer \( i \), with \( 1 \leq i \leq n \)

• Output is an element \( x \) (taken from the input set) such that \( x \) is larger than exactly \( i-1 \) elements of the input set (i.e. \( x \) is the \( i \)-th largest element)
  - Min and Max are special cases (with \( i=1 \) and \( n \), resp.)
Selection in Expected Linear Time

Randomized-Select(A[1..n], i)
  if n == 1 then return A[1]
  q = Randomized-Partition(A[1..n])
  if i == q then return A[q]
  if i < q then return Randomized-Select(A[1..q-1], i )
  else return Randomized-Select(A[q+1..n], i - q)

• What is the worst case runtime? ____
• What is expected runtime?
  ▪ Randomized-Partition selects from n possible pivots uniformly
  ▪ Pretend the “worst case” always happens in the conditional
  ▪ \( E[T(n)] \leq O(n) + \left( \frac{1}{n} \right) \sum_{q=1}^{n} T(\max(q-1,n-q)) \)
  ▪ Prove \( E[T(n)] = O(n) \) by induction… see CLRS for details
Selection in **Worst Case Linear Time**

- A high level description of algorithm \texttt{Select}(A[1..n], i):
  1. Divide input array \( A \) into \( n/5 \) groups of 5 elements each
  2. Find the medians of each of the small (5 element) groups. Observe this takes constant time per group, and is thus an \( O(n) \) time operation. Call the set of all \( n/5 \) of these medians \( X \).
  3. Recursively call \texttt{Select}(X,n/2) to find the median of \( X \). Call this median \( x \) (it is a “median of medians”).
  4. Partition \( A \) using \( x \) as the pivot value. Let \( q \) be the position of the pivot in the partitioned array (i.e. \( x \) is the \( q \)-th largest, and we have \( A[q] == x \) after partitioning). This takes time \( O(n) \).
  5. If \( i == q \) then return \( x \). Otherwise, call select recursively:
     - if \( i < q \) return \texttt{Select}(A[1..q-1], i)
     - else return \texttt{Select}(A[q+1..n], i - q)

- See CLRS for details of the runtime analysis
  - I will only sketch this in class
Basic Data Structures

• Stacks
  ▪ LIFO – Last In First Out
  ▪ Constant time per “push” or “pop” operation

• Queues
  ▪ FIFO – First In First Out
  ▪ Constant time per enqueue / dequeue operation

• Priority Queues
  ▪ Highest Priority at any given time gets out first
  ▪ Using heaps, $O(\log n)$ time per operation

• (Doubly) Linked Lists
  ▪ Takes $O(n)$ time to find an item, constant time to insert new ones

• Trees
• (Directed) Graphs
Hash Tables

• Generalized notion of arrays
  ▪ Let $U$ denote the set of all data keys (the “universe”)
  ▪ Let $T[0..m-1]$ be an array (or “table”) of size $m$
  ▪ $n$ distinct keys $k_1, \ldots, k_n \in U$ must be stored in the table

• “Direct Addressing” is natural, but inefficient
  ▪ Requires $m = |U|$, so storage requirements are $O(|U|)$

• Hashing increases storage efficiency when $n << |U|$  
  ▪ A hash function $h$ maps $U$ into $\mathbb{Z}_m = \{0, \ldots, m-1\}$  
    • For this to be useful, we require that $m < |U|$
  ▪ Thus key $k$ can be stored in $T[h(k)]$
  ▪ Since $|U| > m$, there must be some collisions
Collision Resolution by Chaining

• Make each table entry a linked list
  ▪ Key k is present if it is in the linked list \( T[h(k)] \)
  ▪ To Insert k in the table, add it to the front of \( T[h(k)] \)
  ▪ To Find key k, search the linked list in \( T[h(k)] \)
  ▪ To Delete key k, Find it and then remove it from the linked list

• What is the worst case runtime?
  ▪ \( h \) is constant time (it is independent of \( n \))
  ▪ Insert is constant time, Find is \( O(n) \) if \( O(n) \) keys collide

• What is the expected runtime if keys hash uniformly?
  ▪ Define the load factor \( \alpha \) to be \( n/m \)
  ▪ Let \( n_j \) denote the length of the list \( T[j] \)
    ▪ Clearly \( n = \sum_{j=0}^{m} n_j \)
  ▪ Easy to see that \( E[n_j] = \alpha \) for \( j = 0, ..., m-1 \)
  ▪ Expected search time is thus \( \Theta(1+\alpha) \)
  ▪ See detailed CLRS analysis, to be sketched in class
Obtaining Uniform Hashing

• If keys are uniformly distributed in [0,1):
  ▪ \( h(k) = \lfloor mk \rfloor \) (could handle other intervals easily too)

• Other ideas (must choose the constants carefully):
  ▪ Division method: \( h(k) = k \mod m \)
  ▪ Multiplication method: \( h(k) = \lfloor m (kA \mod 1) \rfloor \)

• A general method: Universal Hashing
  ▪ Define a hash family \( \mathcal{H} \) containing many hash functions
  ▪ Fix keys \( k_1, k_2 \), then choose a function \( h \in \mathcal{H} \) at random
  ▪ We say \( \mathcal{H} \) is universal if \( \Pr\{h(k_1) = h(k_2)\} = 1/m \)
  ▪ Read CLRS 11.3.3
Designing Universal Hash Families

- Brief intro to number theory required (CLRS 31.3, 31.4)
- Let $p$ be a prime, and $\mathbb{Z}_p = \{0,\ldots, p-1\}$, $\mathbb{Z}_p^* = \{1,\ldots, p-1\}$
- Choose a large prime $p$ such that $p \geq |U| > m$
- Let $\mathcal{H}_{p,m}$ contain functions $h_{a,b}(k)$ where:
  $h_{a,b}(k) = \left\lfloor (ak + b) \mod p \right\rfloor \mod m$
- What can cause this hash function to collide?
  - Let $r = (ak_1 + b) \mod p$, $s = (ak_2 + b) \mod p$
  - For uniform $(a,b) \in \mathbb{Z}_p^* \times \mathbb{Z}_p$ we get uniform $(r,s) \in \mathbb{Z}_p \times \mathbb{Z}_p$
    * Can show this by solving for $(a,b)$ in terms of $(r,k_1)$ and $(s,k_2)$
  - Collision implies $(r = s) \mod m$
  - For random $(r,s) \in \mathbb{Z}_p \times \mathbb{Z}_p$ this happens with probability $1/m$
    * Can show this by counting how many pairs from $\mathbb{Z}_p \times \mathbb{Z}_p$ collide mod $m$
(Linear) Open Addressing

• Technique for $\alpha \leq 1$ only (i.e. $n \leq m$)
• Each table entry contains at most one key
• Insert: If $T[h(k)]$ is empty, insert there. Otherwise look for the next empty spot in $T$, and insert there.
• Find: If $T[h(k)]$ is empty or contains $k$, done. Otherwise, continue searching the next entry until either $k$ or an empty slot is found.
• Performance gets much worse as $\alpha$ approaches 1
• Can use more advanced techniques… see CLRS
Perfect Hashing

- **Worst case** time to Insert and Find is *constant*
- Requires that the set of keys be fixed for all time
  - e.g. the English dictionary
- Uses a “secondary” hash table instead of a linked list (as in chaining), for each of the entries in T
  - $T[j]$ contains a secondary hash table $S_j$
  - Choose a universal hash function $h$ for table $T$
  - Make table $S_j$ of size $m_j = n_j^2$
  - Choose a separate universal hash function $h_j$ for each table $S_j$… keep choosing a new $h_j$ until no collisions occur in $S_j$
  - Can show total expected storage will be $O(n)$, including the space to store the list of $n+1$ universal hash functions
- No collisions occur in secondary tables, so the proper location for any key can be found in two lookups