Given an array of integers, find the length of a longest streak of consecutive integers that occur in it (not necessarily contiguously). For example, if arr = {1,7,9,4,1,7,4,8,7,1}. longestStreak(arr) returns 3, corresponding to the streak {7,8,9} of consecutive integers that occur in the array. Solve using ~~balanced BST~~ hashing, with RT = $O(n$ ~~log n~~$)$, expected time.

```
static int longestStreak(int[] arr) {
    Set<Integer> set = new HashSet<>();
    for ( e: arr) { set.add(e); }
    max ← 0
    for (e: set) {
        if (set.contains (e-1)) continue;
        x = e+1;
        // LI: current streak e, e+1, ..., x-1
                         of length x-e
        while (set.contains (x)) {x++;}
        if (max < x-e) { max = x-e;}
    }
    return max;
```

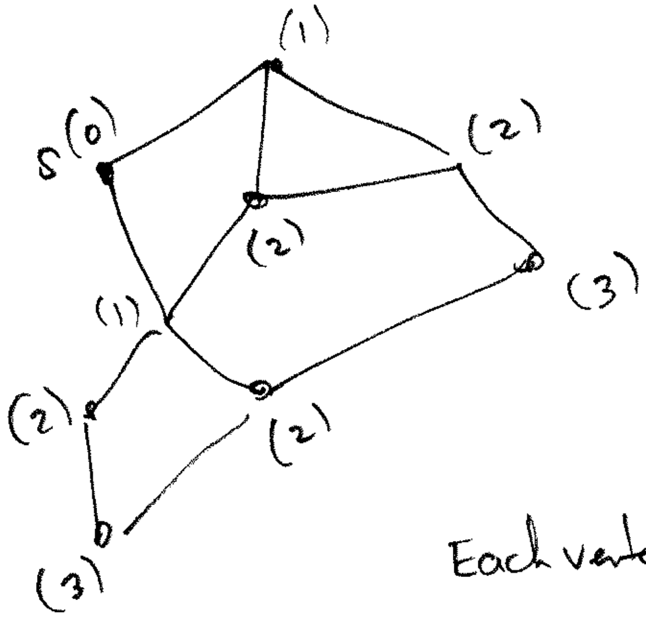
RT analysis:

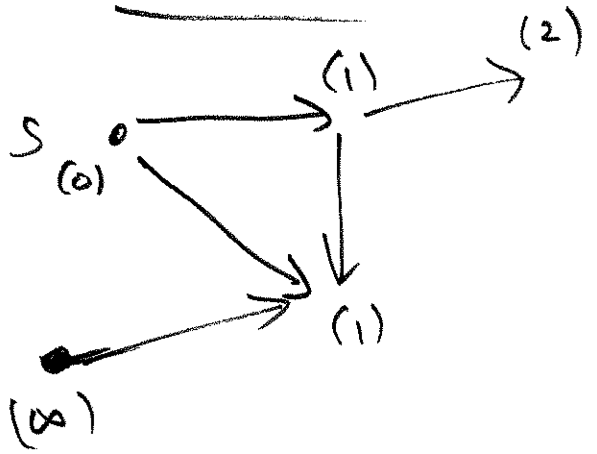
Without considering set operations, RT = O(n).
Each element gives rise to at most 3 set ops.
(a) add(e), (b) contains(e-1), (3) contains(x).
= O(n) expected time.

Breadth-first Search

Traversing nodes of a graph in order of shortest number of hops from a source vertex.

Ex: Undirected   Directed





Each vertex is added to q at most once.

Algorithm:

```
|V| -   for u ∈ $\overset{G}{\underline{V}\cancel{(G)}}$ do   { u.seen ← false , u.dist ← ∞
                                    u.parent ← null }
1 -     s.dist ← 0 , s.seen ← true
1 -     Queue <Vertex> q ← new LinkedList<>() ;
1 -     q.add(s)
        while q is not empty do     // LI: Nodes in q have dist $\underbrace{d, d, \ldots d}_{1 \text{ or more}}, \underbrace{d+1, \ldots d+1}_{0 \text{ or more}}$
|V| -       u ← q.remove()
            for each edge e = (u,v) from u do
2|E| · degree(u)   if v is not seen then
                        v.seen ← true , v.parent ← u, v.dist ← u.dist + 1
O(|V|+|E|)              q.add(v)
```

Graph class: a class to implement graph algorithms. Suppose we want to implement breadth-first search (BFS). First we create a class, say BFSVertex, to store properties of vertices during BFS. This class should implement the Factory interface from the Graph class. For technical reasons, this class has to be static. Factory interface has just one method, make(Vertex u), for creating a node to store properties of vertex u.

```
static class BFSVertex implements Graph.Factory {
    boolean seen;
    Vertex parent;
    int distance;  // distance of vertex from source
    public BFSVertex(Vertex u) {
     seen = false;
     parent = null;
     distance = INFINITY;
    }
    public BFSVertex make(Vertex u) { return new BFSVertex(u); }
}
```

We then create a class BFS for implementing the algorithm. Let the BFSVertex class be a subclass of BFS. We have "BFS extends GraphAlgorithm<BFS.BFSVertex>". We must provide a constructor which takes 2 parameters, the graph on which BFS runs, and a sample BFSVertex that is used by the Factory to make new BFSVertex objects, as needed.

```
public class BFS extends GraphAlgorithm<BFS.BFSVertex> {
    Vertex src;  // source vertex
    public BFS(Graph g) {
     super(g, new BFSVertex(null));
    }
```

To iterate over the vertices of a graph g:

```
// implicit iterator
for(Vertex u: g) { ... }
```

```
// explicit iterator
Iterator<Vertex> iter = g.iterator();
while(iter.hasNext()) {
    Vertex u = iter.next();   ...
}
```

To iterate over the edges incident to vertex u in undirected graph g, or the outgoing edges of u, if g is directed:

```
for(Edge e: g.incident(u)) {
    Vertex v = e.otherEnd(u);   ...
}
```

```
Iterator<Edge> iter =
    g.incident(u).iterator();
while(iter.hasNext()) {
    Edge e = iter.next();    ...
}
```

To access/modify the fields that store properties of a vertex u:

```
Pseudocode:
src.distance ← 0
u.seen ← false
```

```
Java code:
get(src).distance = 0;
get(u).seen = false;
```

Breadth-first search (BFS):

```
// Pseudocode
BFS(G=(V,E), src):
   for vertex u in G do
      u.seen ← false
      u.parent ← null
      u.distance ← ∞
   src.distance ← 0


   q ← new queue of vertices
   q.add(src)
   src.seen ← true

   while q is not empty do
      u ← q.remove()
      for edge e=(u,v) incident to u do
         if not v.seen then
            v.seen ← true
            v.parent ← u
            v.distance ← u.distance + 1
            q.add(v)
```

```java
// Java code for BFS using Graph class
BFS b = new BFS(g, new BFSVertex(null));
b.bfs(src);

public void bfs(Vertex src) {
      this.src = src;
      for(Vertex u: g) {
         get(u).seen = false;
         get(u).parent = null;
         get(u).distance = INFINITY;
      }
      get(src).distance = 0;

      Queue<Vertex> q = new LinkedList<>();
      q.add(src);
      get(src).seen = true;

      while(!q.isEmpty()) {
         Vertex u = q.remove();
         for(Edge e: g.incident(u)) {
            Vertex v = e.otherEnd(u);
            if(!get(v).seen) {
               get(v).seen = true;
               get(v).parent = u;
               get(v).distance = get(u).distance + 1;
               q.add(v);
            }
         }
      }
   }
```

Classes defined in the Graph class and their methods:
Vertex: a class to represent the vertices of a graph.

Operation	**Meaning / Purpose / Usage**
`Vertex(int u)`	Create a vertex named u
`int getIndex()`	Index in which vertex is stored in array of adjacency lists storing graph
`int inDegree()`	Number of incoming edges from vertex
`int outDegree()`	Number of outgoing edges from vertex

Edge: a class to represent the edges of a graph.

Operation	**Meaning / Purpose / Usage**
`Edge(Vertex u, Vertex v, int w, int n)`	Create edge (u,v) of weight w, named n
`Vertex otherEnd(Vertex u)`	One end of edge is u; return other end of edge
`Vertex fromVertex()`	Vertex from which an edge originates
`Vertex toVertex()`	Vertex on which edge lands
`int getWeight()`	Weight of edge
`int setWeight(int nw)`	Set weight of edge to nw. Old weight of edge is returned

Graph: a class to represent undirected and directed graphs, using adjacency lists.

Operation	**Meaning / Purpose / Usage**
`Graph(int n)`	Constructor to create an undirected graph with n vertices and no edges
`Graph(int n, boolean dir)`	Constructor to create graph, dir=true for directed graph
`Edge addEdge(int u, int v, int w)`	Add edge (u,v) with weight w
`int size()`	Number of vertices in graph
`int edgeSize()`	Number of edges in graph
`boolean isDirected()`	Is the graph directed?
`void reverseGraph()`	Reverse the edges of a graph (must be directed)
`Iterator<Vertex> iterator()`	Iterator to go through the vertices of graph, remove( ) not supported
`Graph readGraph(Scanner in)`	Read an undirected graph from in
`Graph readDirectedGraph(in)`	Read a directed graph from in
`Vertex getVertex(int n)`	Vertex named n in the graph

GraphAlgorithm<V extends Factory>: a class that is extended by graph algorithms.

Operation	Meaning / Purpose / Usage
`GraphAlgorithm(Graph g, Factory vf)`	Constructor that sets up storage to run a graph algorithm on graph g, given a factory to make nodes that store properties of vertices
`V get(Vertex u)`	Get the node that stores properties of vertex v
`V put(Vertex u, V value)`	Set node storing v's properties to value, returns old value of node

Factory: an interface to be implemented by classes that store properties of vertices in graph algorithms.

Operation	Meaning / Purpose / Usage
`Factory make(Vertex u)`	Create a node to store properties of vertex u

AdjList: a class to store adjacency list of graph.

Operation	Meaning / Purpose / Usage
`AdjList adj(Vertex u)`	Adjacency list object of vertex u
`AdjList adj(int n)`	Adjacency list object of vertex named n

Store<V extends Factory>: a class to store properties of vertices in a graph, using a parallel array.

Operation	Meaning / Purpose / Usage
`Store(Factory vf)`	Sets up store with an array to store node properties (V)

ArrayIterator<T>: a class that implements an iterator on arrays and subarrays.

Operation	Meaning / Purpose / Usage
`ArrayIterator(T[ ] arr)`	Create an iterator for all elements of arr[0..arr.length-1], no remove( )
`ArrayIterator(T[ ] arr, int start, int end)`	Create an iterator for all elements of arr[start..end], no remove( )

Timer: a class to calculate running time of a program fragment, approximately.

Operation	Meaning / Purpose / Usage
`Timer( )`	Create timer
`void start( )`	Start or restart timer
`Timer end( )`	End timer
`long duration( )`	Elapsed time from start to end in milliseconds
`long memory( )`	Approximate memory used in bytes
`String toString( )`	Called by System.out.print(timer) to print timer statistics