**Node**

This class represents a node in the graph.

Fields:

* int id – id of node.
* int weight – weight of node.
* EdgeDLL edges – doubly linked list of edges emanating from node.
* Neighborhood neighborhood – neighborhood of node.

Methods:

* Node(int id, int weight) – Creates a new node object, given its id and its weight. O(1).
* int getId() – Returns the id of the node. O(1).
* int getWeight() – Returns the weight of the node. O(1).

**Edge**

This class represents an edge emanating from a node in the graph.

Fields:

* int neighborNodeId – id of the node to which edge connects.
* DLLNode<Edge> reciprocalEdge – reference to reciprocal edge.

Methods:

* Edge(int neighborNodeId) – Creates a new edge object, given id of node to which it connects. O(1).

**Neighborhood**

This class represents a neighborhood of a node in the graph.

Fields:

* int nodeId – id of node of neighborhood.
* int weight – weight of neighborhood.
* int heapIndex – index of neighborhood in max neighborhood heap.

Methods:

* Neighborhood(Node node) – Creates a new neighborhood object, given its node. O(1).

**DLLNode**

This class represents a node in a doubly linked list.

Fields:

* DLLNode prev – reference to previous dll node.
* T info – info of dll node.
* DLLNode next – reference to next dll node.

Methods:

* DLLNode(T item) – Creates a new dll node object, given its info. O(1).

**DLL**

This class represents a doubly linked list.

Fields:

* DLLNode<T> sentinel – sentinel of dll.
* int length – length of dll.

Methods:

* DLL() – Creates a new dll object. O(1).
* DLLNode<T> insert(T info) – Adds a new dll node containing info to start of dll and returns added dll node. O(1).
* void delete(DLLNode<T> dllNode) – Deletes dll node, given reference. O(1).

**NodeDLL** (**extends DLL<Node>)**

This class represents a doubly linked list of node objects.

Methods:

* DLLNode<Node> getDLLNode(int nodeId) – Returns dll node containing info node with given id, or null if no such dll node exists. O(n) – may iterate over entire list.
* Node getNode(int nodeId) – Returns info node contained in dll node with given id, or null if no such node exists. O(n) – may iterate over entire list.
* void delete(int nodeId) – Delete dll node containing info node with given id, if exists. O(n) – may iterate over entire list.

**EdgeDLL (extends DLL<Edge> implements Iterable<Edge>)**

This class represents a doubly linked list of edge objects.

Methods:

* Iterator<Edge> iterator() – Returns iterator of edge dll. O(1).

**NodeHashTable**

This class represents a hash table of node objects.

Fields:

* NodeDLL[] hashTable – array of node dlls of hash table.
* int m – m of modular hash function of hash table.
* int a – a of modular hash function of hash table.
* int b – b of modular hash function of hash table.

Methods:

* NodeHashTable(int size) – Creates a new hash table object with random modular hash function, given its size. O(1).
* int getHashValue(int i) – Returns hash value of given integer. O(1).
* void insert(Node node) – Inserts node. O(1).
* Node get(int nodeId) – Returns node with given id, or null if no such node existed. O(1) - Expected.
* void delete(int nodeId) – Deletes node with given id, if exists. O(1) - Expected.

Since size of hash table is same as the initial number of nodes in the graph it load factor is less than or equal to one, therefore expected length of each linked list in the hash table is O(1).

**MaxNeighborhoodHeap**

This class represents a max binary heap of neighborhood objects. Heap order determined by neighborhood weight.

Functions in this class assume that heap is not empty (size>0) and indexes given are legal (less than size).

Fields:

* Neighborhood[] neighborhoods – array of heap.
* int size – size of heap.

Methods:

* MaxNeighborhoodHeap(Neighborhood[] neighborhoods) – Creates a new max binary heap of neighborhood objects, given array of neighborhood objects. O(n) – heapify array from top to bottom.
* void swap(int i, int j) - Swaps neighborhood objects, given their indexes. O(1).
* void heapifyDown(int i) - Heapify down neighborhood object, given its index. O(log n) – may heapify down height of tree.
* void heapifyUp(int i) - Heapify up neighborhood object, given its index. O(log n) – may heapify up height of tree.
* void changeKey(int i, int delta) - Changes weight of neighborhood object by delta and heapifies, given its index and change in weight. O(log n) – may heapify height of tree.
* void delete(int i) - Deletes neighborhood object, given its index. O(log n) – switches neighborhood with last neighborhood which may then be heapified height of tree.
* int getMaxNeighborhoodNodeId() - Returns node id of max neighborhood weight. O(1).

**Graph**

Functions in this class assume that user will not cause insertion of duplicated.

Fields:

* int numNodes – number of nodes in graph.
* int numEdges – number of edges in graph.
* NodeHashTable nodes – hash table of nodes in graph.
* MaxNeighborhoodHeap maxNeighborhoodHeap – max binary heap of neighborhoods of nodes in graph.

Methods:

* Graph(Node[] nodes) - Initializes the graph on a given set of nodes, by creating a hash table of nodes and a max binary heap of their neighborhoods. The created graph is empty, i.e. it has no edges.

Complexity –

Creating node hash table and inserting n nodes into it. O(n).

Creating array of neighborhoods of nodes and creating a max heap out of it. O(n).

Total = O(n).

* Node maxNeighborhoodWeight() - This method returns the node in the graph with the maximum neighborhood weight, or null if graph is empty.

Complexity –

Getting node id of neighborhood with max weight. O(1).

Getting node in hash table. O(1) – Expected.

Total = O(1) – Expected.

* int getNeighborhoodWeight(int node\_id) - Given a node id of a node in the graph, this method returns the neighborhood weight of that node, or -1 if no such node exists.

Complexity –

Getting node in hash table and returning its neighborhood weight, if node exists. O(1) – Expected.

Total = O(1) – Expected.

* void changeNeighborWeight(Node node, int delta) - Given a node of a node in the graph, this method changes the neighborhood weight of that node by delta.

Complexity –

Changing weigh of neighborhood in max heap. O(log n).

Total = O(log n).

* boolean addEdge(int node1\_id, int node2\_id) - This function adds an edge between the two nodes whose ids are specified. If one of these nodes is not in the graph, the function does nothing. The two nodes must be distinct; otherwise, the function does nothing. Returns true if edge added, otherwise returns false.

Complexity –

Getting two nodes in hash table. O(1) – Expected.

Adding two edges to edge dlls of nodes. O(1).

Updating two nodes neighborhood weight. O(log n).

Incrementing numEdges by 1.

Total = O(log n).

* boolean deleteNode(int node\_id) - Given the id of a node in the graph, deletes the node of that id from the graph, if it exists. Returns true if node deleted, otherwise returns false.

Getting node in hash table. O(1) – Expected.

For each edge of node (d), delete reciprocal edge from neighbor, O(1), update neighbor height, O(log n), and decrease numEdges by 1. O(d log n).

Delete node neighborhood from max heap, O(log n), delete node from hash table, O(1) – Expected, and decrease numNodes by 1. O(log n).

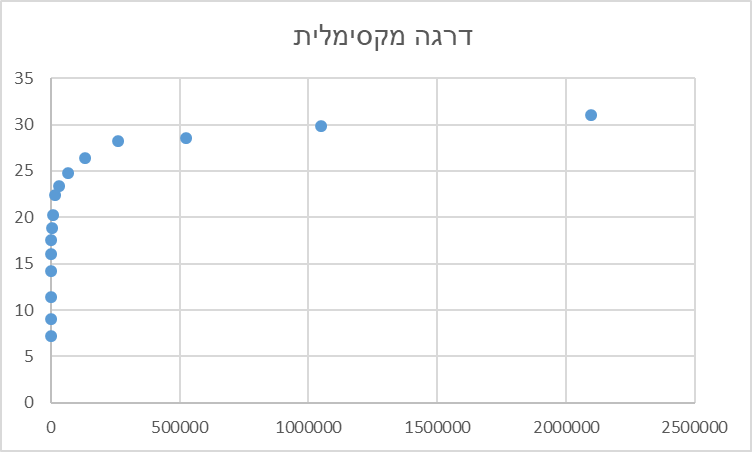
Total = O((d+1) log n).

* int getNumNodes() – Return number of nodes in graph. O(1).
* int getNumEdges() – Return number of edges in graph. O(1).

**ניסוי**

ממוצע על פני 5 הרצות – מעוגל.

|  |  |  |
| --- | --- | --- |
| מספר סידורי i | n | דרגה מקסימלית בגרף |
| 6 | 64 | 7 |
| 7 | 128 | 9 |
| 8 | 256 | 11 |
| 9 | 512 | 14 |
| 10 | 1024 | 16 |
| 11 | 2048 | 18 |
| 12 | 4096 | 19 |
| 13 | 8192 | 20 |
| 14 | 16384 | 22 |
| 15 | 32768 | 23 |
| 16 | 65536 | 25 |
| 17 | 131072 | 26 |
| 18 | 262144 | 28 |
| 19 | 524288 | 29 |
| 20 | 1048576 | 30 |
| 21 | 2097152 | 31 |

****

נראה שהדרגה המקסימלית של צומת בגרף לא מכוון בעל n צמתים וn קשתות שנבחרו באופן אקראי גדל באופן "לוגרתמי" ביחס לn.

הקשר שבין הניסוי שלנו לבעיית הballs into bins הוא שניתן לראות את הניסוי שלנו באופן הבא:

* כל צומת במערך ייוצג על ידי bin. סה"כ n bins.
* כל בחירה אקראית של קשת מתוך ה קשתות האפשריות היא כמו להניח כדור בbins של הצמתים המעורבים בקשת.
* סה"כ אנחנו מכניסים 2n כדורים לn bins, כאשר בחירת הזוגית היא אקראית – אך ללא חזרות.

ממה שקראנו בhttps://en.wikipedia.org/wiki/Balls\_into\_bins\_problem, התוצאות האמפיריות שלנו מתיישבות חלקית אם מה שניתן לצפות. מצד אחד קיבלנו גידול "לוגרתמי" בדומה לגידול בבעיית הball into bins מצד שני קצב הגידול בניסוי שלנו הוא גדול יותר. אנחנו חושבים שאולי הסיבה להבדל היא שההכנסה אצלנו שונה מההכנסה בבעייה זו. אצלנו אנחנו בוחרים זוגות אקראיים מ. זו אינה בחירה אקראית על המספרים 1-n אלא בחירה בכל פעם להכניס כדורים באינדקסים ללא חזרות.