Map, hash table and skip list

1 Maps and dictionary

A dictionary or map is a collection of pairs *(key, values)* that is searchable. The main operations of a map are inserting an item, deleting an item, and finding an item. The map do not allow for multiple items to have the same key. Given a map M it is possible to access the value of the item with key k with $M[k]$. We can insert an item using $M[k] = v$. We can delete an item with `del M[k]`. `len(M)` return the number of items in the dictionary. The default iteration `__iter__` for a map generate the sequences of *keys* of the map, i.e. it is equivalent to `for k in M`. `k in M` return True if the key is in the map. `M.get(k, d=None)` return the value corresponding to key k if it exist else it returns None. `M.setdefault(k, d)` return `M[k]` if it exist else set `M[k]=d` and return `d`. `M.pop(k, d)` remove the item with key `k` and return its value; if $k \notin M$ return d or raise a `KeyError` if d is None. `M.popitem()` remove an arbitrary pair (k, v) from M and return it. `M.clear` empty the map. `M.keys()` and `M.values()` return respectively the set of keys and the set of values in the map. `M.items()` return a set of pairs in the map. `M.update(D)` will set `M[k]=v` for each pair $(k, v) \in D$. `M1==M2` return `True` if all `M1` and `M2` have identical pair-value associations.

MutableMapping abstract base class. The class `MutableMapping` provide a concrete implementation of all the method describe in above except for the methods `__getitem__`, `__setitem__`, `__iter__`, `__len__`, and `__delitem__`. This means that a class that inherit from `MutableMapping` should implement these methods, else a *NotImplementedError* should be raise.

The Map class. Before implementing the class we need an `_Item` to represent the objects in the map.

```
class Item:
    def __init__(self, key, value=None):
        self._key = key
        self._value = value

    def key(self):
        return self._key

    def value(self):
        return self._value
```

Now we can implement the class Map. The class Map needs a `is_empty()` method that return True if the map is empty, a `get(`k`)` method that return the value corresponding to kek k, and a `setdefault(`k`, `d`)` method that returns the value corresponding to the key k if it is in the map, else it set `M[`k`]=`d and return d.

1.1 Implementing Map with a list.

We can implement Map with an unsorted list. We stock the element of a Map is a list S in an arbitrary order. We need to implement 7 methods for the class Map: `__setitem__` set a key to a particular value and

```python
class Map(MutableMapping, Item):
    def is_empty(self):
        return len(self) == 0

    def get(self, k, d=None):
        try:
            return self[k]
        except KeyError:
            return d

    def setdefault(self, k, d=None):
        if self.get(k) is not None:
            return self.get(k)
        else:
            self[k] = d
            return d
```

Listing 1: The Map class.

return nothing, `__getitem__`, `__len__`, `__iter__` that provide an iterator on the keys, `__items__` that provide an iterator on the items, `__contains__`, and `__delitem__`.

```python
class ListMap:
    def __init__(self):
        self._T = []

    def __getitem__(self, k):
        for item in self._T:
            if k == item._key:
                return item._values
        raise KeyError(k)

    def __setitem__(self, k, v):
        for item in self._T:
            if k == item._key:
                item._value = v
                return
        self._T.append(Item(k, v))

    def __len__(self):
        return len(self._T)

    def __contains__(self, k)
        try:
            self[k]
        except KeyError:
            return False
        return True

    def __iter__(self):
        for item in self._T:
            yield item.key()

    def __items__(self):
        for item in self._T:
            yield (item.key(), item.value())

    def __delitem__(self, k):
        for i, item in enumerate(self._T):
            if k == item.key():
                self._T.pop(i)
                return
        raise KeyError(k)
```

Performance of the Map implementation with an unsorted list Inserting can be done in $O(1)$ since we can insert an item at the end of the list using `append`, but because we need if the key is already in the Map, the operation takes $O(n)$. Searching and deleting takes $O(n)$-time since we need to loop in all the items of the Map in the worst case. The implementation with the unsorted list is practical only for Map of small size.

2 Hash Tables

A map support the abstraction of using keys to access values with the syntax $M[K]$. This is similar to the representation of a lookup table. A a mental warm-up, we consider restricted case where the Map M have n elements with integer keys $0, 2, ..., N-1$ for $N \geq n$. For example, consider the *Map* $\{1\colon D,\ 3\colon Z,\ 6\colon C,\ 7\colon Q\}$. Basic map operations `__setitem__`, `__getitem__`, and `__delitem__` can be done in $O(1)$ in worst case. There

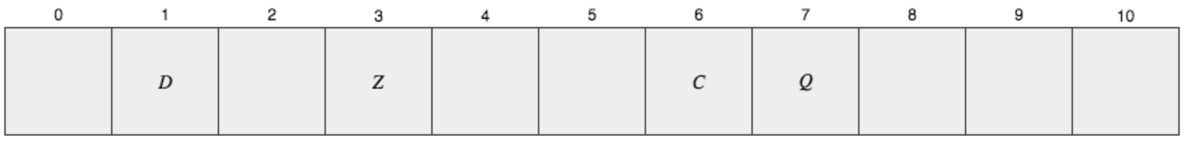

Figure 1: Simple lookup table.

is two challenge in extending this case to the more general context of a Map. First, we do not want to use a table of size N when N is much larger then n. Secondly, we don't require the keys to be integer. The novel idea is to use a ***hashing function*** to map the keys to the position of there values in the table. Ideally, the keys are well distributed across $0, 1, ..., N-1$. In practice it is possible for 2 distinct keys to map to the same index in the table. Thus we represent the Map as a table of container.

A **hashing function** h is a mapping $h\colon k \mapsto [0, N-1]$, i.e. h maps a key to an integer between 0 and $N-1$. For example $h(k) = k \mod N$ is a valid hashing function if the keys are integers. The integer $h(k)$ is called the hashing value of the of the key k. An **hashing table** for a type of key consist of a hashing function h and a table size N. When implementing a Map with an hashing table, the goal is to store item (k, v) at index $i = h(k)$. When two items point to the same entry in the a hashing table, a **collision** happen. This happen if $k_1 \neq k_2$, but $h(k_1) = h(k_2)$. We will see later how to resolve this problem.

A hashing function is generally a composition of two function: a first function, called the *hash code* maps the key to an integer $h_1\colon k \mapsto \mathbb{N}$ and a second function, called the *compression function* maps the output of the first mapping to an index in the table $h_2\colon \mathbb{N} \mapsto [0, ..., N-1]$. The goal of the hashing function $h(k) = h_2(h_1(k))$ is to distribute the keys in a random-like manner.

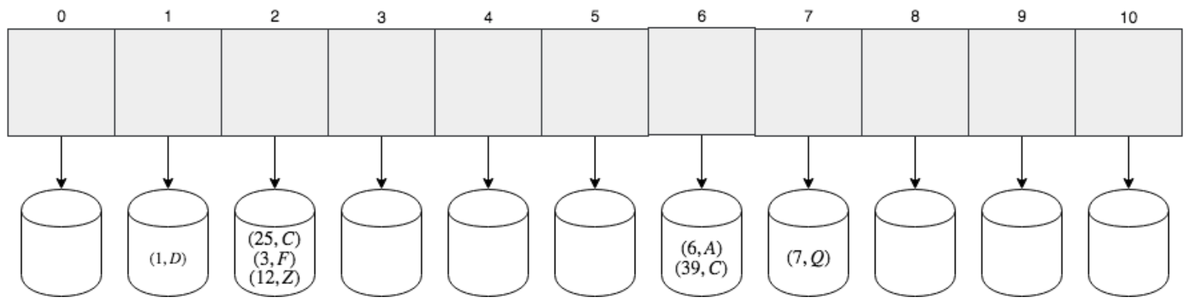

Figure 2: A bucket array of capacity 11.

```
class HashMap(Map):
    def __init__(self, cap=11, p=109345121):
        self._T = cap * [None]
        self._n = 0
        self._prime = p
        self._scale = 1 + randrange(p-1)
        found = False
        while not found:
            self._scale = 1 + randrange(p-1)
            if not self._scale % p == 0:
                found = True
       self._shift = randrange(p)
       self._mask = cap
```

Listing 2: The Map class.

3 Search Tree

3.1 Binary search tree

A binary search tree is a binary tree storing a pair *(key, value)* in each of its node. Binary search trees respect the following property: given 3 nodes u, v, and w such that u is in the left sub-tree of v and w is in the right sub-tree of v, then $key(u) < key(v) < key(w)$. A inorder traversal of a tree (*left node-root-right node*) visit the keys in increasing order. The most fundamentals methods of the BST are `__getitem__`, `__setitem__`, and `__delitem__`.

Other methods a very useful to facilitate the navigation it the tree. The method `first()` return the position containing the least key. The method `last()` return the element containing the greatest key. The method `before(p)` returns the position with the greatest key that is lower then the key of position p. The method `after(p)` return the position containing the smallest key that is greater then the key of position p. The first position can be found by doing walk from the root following the left childs of the tree as long as one exist. Similarly We can reach the last position by starting at the root and continuing to the right child until we reach a leaf.

Search. To find a key k, we follow a descending path starting at the root. The next node to visit depend on the comparison of the current node key with k.
Insertion In order to insert a new node (k, v) in the tree, we first search for the key k using `TreeSearch`. Let w be the node were the search ended. If the there is a node with the key k, i.e. w is not None, the value

```
def first(self):
    p = self.root()
    while p is not None:
        p = self.left(p)
    return p

def last(self):
    p = self.root()
    while p is not None:
        p = self.right(p)
    return p
```

Listing 3: method first

```
def _subtree_first_position(p):
    """Return the first position of the subtree of which p is the root"""
    while left(p) is not None:
        p = left(p)
    return p

def after(p):
        if right(p) is not None:
            p = right(p)
            return _subtree_first_position(p)
        else:
            ancestor = parent(p)
            while not ancestor == None and p == right(ancestor):
                p = ancestor
                ancestor = parent(p)
            return ancestor
```

Listing 4: Computing the successor of a position in a binary search tree.

of this node is set to v. Else, a new node is added at the right position of w if $w.key() < k$ or at the left if $w.key() > k$.

```
def TreeInsert(T, k, v):
    p = TreeSearch(k)
    if k == p.key():
        p.element()._value = v
        return
    else:
        item = _Item(k, v)
        if k < p.key():
            _add_left(p, item)
        else:
            _add_right(p, item)
```

```
def _subtree_last_position(p):
        """Return the first position of the subtree of which p is the root"""
        while right(p) is not None:
        p = right(p)
        return p

def before(p):
        if left(p) is not None:
            p = right(p)
            return _subtree_last_position(p)
        else:
            ancestor = parent(p)
            while ancestor is not None and left(ancestor) == p:
                p = ancestor
                ancestor = parent(p)
            return ancestor
```

Listing 5: Computing the successor of a position in a binary search tree.

```
def TreeSearch(T, k, p):
    if k == p.key():
        return p
    elif k < p.key() and T.left(p) is not None:
        return find(T, k, T.left(p))
    elif k > p.key() and T.right(p) is not None:
        return find(T, k, T.right(p))
    else:
        return None
```

Listing 6: Find and return the position with the key `k`.

```
def TreeDelete(T, p):
    node = p._node
    child = node._left if node._left is not None else node._right
    if child is not None:
        child._parent = node._parent
    if node is T.root():
        self._root = child
    else:
         parent = node._parent
         if node is parent._left:
             parent._left = child
         else:
             parent._right = child
     self._size -= 1
     node._parent = node
     return node._element

def TreeDelete(T, k):
    p = TreeSearch(k)
    if not p._node._element.key() == k:
        raise KeyError
    if T.num_children(p) < 2:
        parent = T.parent(p)
        T._delete(p)
    else:
        next_position = T.after(p)
        p._node._element = next_position._node._element
        parent = T.parent(next_position)
        TreeDelete(T, next_position)
    T._rebalance_delete(parent)
```

3.2 Balanced Search Trees

The best case for the binary search tree is when the tree is balanced, i.e. his height is $logn$. Thus we now consider techniques to keep the tree balanced after each operation. The tree will be rebalanced when searching, deleting, and inserting. There is 3 balanced binary tree structure: AVL, Splay and RedBlack. They distinguish themselves by the way they implement the rebalancing ops.

- Scenario when the tree is rebalanced with `_rebalance_access(p)`
  - The item with key k is access with `M[k]` ; `_rebalance_access(p)` is called where p is either a position with the node having key k or the position where the search ended.
  - The private method `p = self._find_position(k)` is called. Then `_rebalance_access(p)` is called.
  - Insertion with `M[k]=v` and k is already in the tree. Then `_rebalance_access(p)` is called where p is the position with key k.
  - Deletion of item with key k when k is not in the tree. `_rebalance_access(p)` is called where p is the position where the search ended.
- Scenario when `_rebalanced_insert(`p`)` is called
  - Insertion `M[k]=v` and k in not in the tree, then `_rebalanced_insert(`p`)` is called where p is the new node's position
- Scenario when `_rebalanced_delete(`p`)` is called
  - Deletion `del M[k]` and k is in the tree, then call `_rebalanced_delete(`p`)` where p is the position of the parent's position of the deleted node.

3.2.1 AVL Tree

AVL tree are binary tree with the property that for each internal node v, the height of the children of v can differ of maximum 1. By convention, the height of external nodes is equal to 1. The height of an internal node is the maximum height of is children plus 1. The height of an AVL tree storing n keys is $O(\log n)$.

4 Splay Tree

Splay tree do not have logarithm limit on there height. The operation move-to-root, called **splaying** is called at each insertion, suppression, and search. The most frequently accessed elements stay closer to the root reducing the there access time. Given a node x in a BST T we splay x by moving it to the root of T with a sequence of restructuration. The restructuring operation to exacute in order to move x depends on the relative position of x, is parent y and its grand-parent z. There is 3 cases to consider:

Zig-zig The node x and its parent y are both left children or both right children. We push x toward the root by making y achild of x and z a child of y.

Zig-zig One of the nodes x and y is a left child and the other is a right child. We push x to the root by making both y and z childrens of x.

Zig The node x do not have grand-parent. In this case, we do one rotation to bring up x compared to y by making y a child of x.

The cost of splaying is in $O(h)$ where h is the height of the tree.

When and which node to splay. When **Searching** for the key k, if the key k is found at position p, we splay p. If the key k is not found, we splay the position where the search ended. When inserting a key k we splay the newly created node. When **deleting** a key k we splay the position of the parent of the deleted node.

```
def _splay(self, p):
    while not p == self.root():
        parent = self.parent(p)
        grand_parent = self.parent(parent)
        if grand_parent is None:
            self._rotate(p)
        elif (parent == self.left(grand_parent)) == (p == self.left(parent)):
            self._rotate(parent)
            self._rotate(p)
        else:
            self._rotate(p)
            self._rotate(p)
```

5 $(2,4)$ Tree

A (2, 4) tree is a sorted tree whose internal node possesses between 2 and d childs and store a maximum of $d-1$ items (k_i, o_i). For a node with children $v_1, ..., v_d$ storing the keys $k_1, ..., k_{d-1}$, the key of the subtree of v_1 are smaller than k_1, the key of the subtree of v_i are between k_{i-1} and k_i, and the key of the subtree v_d are higher then k_{d-1}. The leaf do not store any element. In a $(2,4)$ tree, all the external nodes have the same depth.

Proposition 1. *The height of a* $(2,4)$ *tree is in* $O(\log n)$.

Searching in a $(2,4)$ tree thus take $O(\log n)$ time.
Insertion We insert a new element (k, o) at the leaf reached when searching the key k. If inserting the new item cause a node to have more than 3 keys, we need to split the tree.

6 Graph

A graph is a pait (V, E) where V is a set of nodes and E is a set of edges. Edges are pairs of nodes (u, v) indicating there is an arc between node u and node v. The nodes and edges are position storing elements. There is two types of edges: directed and undirected. If $e = (u, v)$ is a directed edges it means it is possible to go from u to v, but it is not possible to go from v to u. **Directed graphs** contain only directed edges and **undirected graphs** contain only undirected edges. If an edge $e = (u, v)$ connect the nodes u and v we say that u and v are **endpoints** or **end vertices** of e. We say that node u and v are **adjacent** if there is an edge whose end vertices are u and v. An edge is said to be incident to a node if the node is one of the edge endpoint. We say that two edge are adjacent if they join the same node. The degree of a node is the number of adjacent edge of this node with the particularity that self-loop add 2 instead of 1. Two edge are parallel if they connect the same end node. A **path** is a sequence of nodes and edges starting and ending with a node. We sat that a path is **simple** if its edges and nodes are not repeated. A graph is said to be connected for any two nodes there exist a path connecting them. A directed graph is strongly connected if for any two nodes u and v there exist a path from u to v and a path from v to u. A **sub-graph** of a graph $G = (E, V)$ is a graph $H = (\tilde{E}, \tilde{V})$ such that $\tilde{E} \subset E$ and $\tilde{V} \subset V$. A **spanning subgraph** is a subgraph of G that contains all the nodes of G. If G is not connected, the maximal subgraph of G are called the **connected component** of G. A **forest** is a graph without cycle. A **tree** is a connected forest. A **spanning tree** is a spanning graph of a tree, i.e. the spanning graph of a connected graph without cycle. A graph without parallel edges and self-loop is said to be **simple**.

Proposition 2. *Let m be the number of edges in a graph $G = (E, V)$, then*

$$\sum_{v \in V} \deg(v) = 2m$$

Proposition 3. *Let m be the number of edges and n be the number of nodes of a simle undirected graph $G = (E, V)$ (no parallel edges nor self loop), then*

$$m \leq n(n-1)/2$$

If G is directed, then

$$m \leq n(n-1)$$

```python
class Vertex:
    def __init__(self, x=None):
        self._element = x

    def element(self):
        return self._element

    def __hash__(self):
        return hash(id(self))
```

```python
class Edges:
    def __init__(self, u, v, x=None):
        self._origin = u
        self._destination = v
        self._element = x

    def endpoints(self):
        return (self._origin, self._destination)

    def opposite(self, v):
        return self._destination if v == self._origin else self._origin

    def element(self):
        return self._element

    def __hash__(self):
        return hash((self._origin, self._destination))
```

The incidence $I(v)$ maps the node v to a mapping $\cdot : U_v \mapsto E_v$ where U_v is the set of adjacent node of v and E_v is the set of incident edge of v. When the graph is directed, there is an outgoing mapping $I_{out}(v)$ and an ingoing mapping $I_{in}(v)$.

```
class Graph(Vertex, Edges):
    def __init__(self, durected=False):
        self._outgoing = {}
        self._ingoing = {} if directed else self._outgoing

    def vertex_count(self):
        return len(self._outgoing)

    def vertices(self):
        return self._outgoing.keys()

    def edge_count(self):
        total = sum([len(self._outgoing[v]) for v in self.vertices()])
        return total if self.is_directed() else total // 2

    def edges(self):
        edges = set()
        for incident_map in self._outgoing.values():
            edges.updata(incident_map.values())
        return edges

    def get_edge(self, u, v):
        return self._outgoing[u].get(v)

    def degree(self, v, outgoing=True):
        adjacent = self._ougoing if outgoing else self._ingoing
        return len(adj[v])

    def incident_edges(self, v, outgoing=True):
        adjacent = self._ougoing if outgoing else self._ingoing
        for edge in adjacent[v].values():
            yield edge

    def insert_vertex(self, x=None):
        v = self.Vertex(x)
        self._outgoing[v] = {}
        if self.is_directed():
            self._ingoing[v] = {}
        return v

    def insert_edge(self, u, v, x=None):
        e = self.Edge(u, v, x)
        self._outgoing[u][v] = e
        self._incoming[v][u] = e
```

6.1 Depth first search

Depth first search (DFS) is a general technique for traversing a tree. The DFS traversal of a graph visit all the nodes and edges of G, determine if G is connected, find the connected components of G and find a spanning forest of G. Executing DFS on a graph with n nodes and m edges takes $O(n+m)$ time.

```
def DFS(g, u, visited={}):
    for e in g.incident_edges(u):
        v = e.opposite(u)
        if v not in visited:
            visited[v] = e
            DFS(g, v, visited)
```

Proposition 4. *Let G be an undirected graph on which we run DFS from the vertex s. Let H be a connected component of G such that $s \in V_H$. And let E_{DFS} be the edges discovered by the DFS. Then the graph $T = (V_H, E_{DFS})$ is a spanning tree of H.*

Proposition 5. *Let V_{DFS} be the set of node visited by the DFS and E_{DFS} be the set of edges traversed doing the DFS. Then, the graph $H = (V_{DFS}, E_{DFS})$ is a connected subgraph without cycle of G, i.e. H is a spanning tree of G*

Proposition 6. *All the nodes are visited by DFS*

Proposition 7. *Let $G = (V, E)$ be a directed graph. Let $H = (V, E_{DFS})$ be the tree resulting by the DFS started at node $s \in V$. Then H contains a path from s to u for all $u \in V$.*

DFS performance. DFS can be execute in $O(n + m)$ time, where n is the number of nodes and m is the number of edges assuming the following conditions: the method returning all the edges connected to the node v `incident_edges(v)` can be done in $O(deg(v))$ where $deg(v)$ is the number of incident edges of v and `opposite(v)` takes $O(1)$ time.

DFS allows us to solve a number of interesting problems in $O(n + m)$. Find the path between two nodes of G if there exist one. Test if G is connected. Find a spanning tree of G if G is connected. Find the connected components of G. Find a cycle in G or determining that G has no cycle.

If G is directed, the following problem can be solve in $O(n+m)$: Finding a directed path between two nodes, finding all the vertices of G that are accessible from s, testing if G is strongly connected, finding a directed cycle in G.

```
def construct_path(g, u, v, discovered):
    path = []
    if v in discovered:
        path.append(v)
        walk = v
        while walk is not u:
            e = discovered[walk]
            parent = e.opposite(walk)
            path.append(parent)
            walk = parent
        path.reverse()
    return path
```

6.1.1 Finding the connected component of a graph

When a graph is not connected, we can find his connected component. If G is undirected if calling DFS do not return all the nodes, we need to call DFS on a different node.

```
def DFS_complete(g):
    forest = {}
    for u in g.vertices():
        if u not in forest:
            forest[u] = None
            DFS(g, u, forest)
    return forest
```

6.2 Breath First Search

```
def BFS(g, s, discovered):
    level = [s]
    while len(level) > 0:
        next_level = []
        for u in level:
            for e in g.incident_edges(u):
                v = e.opposite(u)
                if v not in discovered:
                    discovered[v] = e
                    next_level.append(v)
        level = next_level
```

Property of BFS

- BFS on undirected graph =¿ $\forall e \in \{e \in G : e \notin E_{BFS}\}$ e is a cross edge.
- BFS on directed graph =¿ $\forall e \in \{e \in G : e \notin E_{BFS}\}$ e is a cross edge or a back edge
- For all node v at level i the path from s to v has i edges and any other path from s to v has at least i edges.
- If $e = (u, v) \notin E_{BFS}$ then $0 \leq level(v) - level(u) \leq 1$