Guía de Trabajos Prácticos número 2
Tipos de datos abstractos fundamentales

[1] [Teoría y Operativos]

a) ¿Cuál es la signatura de la función `insert()` en listas STL?. Diga qué es cada una las variables y cuál es el tipo de retorno. En caso de que haya varias signaturas explique una de ellas.

b) ¿Que retorna el constructor por defecto de lista?

c) ¿Cuál es el tipo de la dereferenciación de un iterator a lista para listas simplemente enlazadas?

d) ¿Cómo es la clase `cell` en listas simplemente enlazadas? ¿Y para listas doblemente enlazadas?

e) Sea `L` una lista conteniendo los elementos $(1, 3, 4, 2, 5, 6)$. Después de aplicar las siguientes líneas

```
list<int>::iterator p,q;
p = L.begin();
q = ++p;
p = L.erase(q);
p++;
q = p;
q++;
```

¿Cuál de las siguientes opciones es verdadera?

- `*p=2, *q=5`.
- `*p=2`, `q` es inválido.
- `*p=4`, `*q=2`
- `*p=4`, `q` es inválido.

f) ¿Puede tener una correspondencia varios valores iguales del dominio, o sea varias claves iguales? (Por ejemplo `M1={(1,2),(1,3)}`) ¿Y varios valores iguales del contradominio? (Por ejemplo `M2={(1,2),(3,2)}`)

g) Sea la correspondencia `M={(1->2),(5->8)}` y ejecutamos el código `int x= M[5]`. ¿Que ocurre? ¿Qué valores toman `x` y `M`? ¿Y si hacemos `x = M[3]`?

h) ¿Cúal es el orden algorítmico del método `find()` para las posibles implementaciones de correspondencias?

i) ¿Cuáles son los tiempos de ejecución para los diferentes métodos de la clase `map<>` implementada con *listas desordenadas* en el caso promedio?
Métodos: `find(key)`, `M[key]`, `erase(key)`, `erase(p)`, `begin()`, `end()`, `clear()`.

j) ¿Qué ventajas o desventajas tendría implementar la clase `pila<>` en términos de *lista simplemente enlazada* poniendo el tope de la pila en el fin de la lista?

[2] [Programación]

a) **[Listas]**

1) *BasicSort*. Escribir una función `void basic_sort(list<int> &L)`, que ordena los elementos de `L` de menor a mayor. Para ello emplear el siguiente algoritmo simple: utilizando una lista auxiliar `L2`, tomar el menor elemento de `L`, eliminarlo de `L` e insertarlo al final de `L2` hasta que `L` este vacía. Luego insertar los elementos de `L2` en `L`.
2) *SelectionSort*. Escribir una función `void selection_sort(list<int> &L)`, que ordena los elementos de `L` de menor a mayor. Para ello debe tomarse el menor elemento de `L` e intercambiarlo (swap) con el primer elemento de la lista. Luego intercambiar el menor elemento de la lista restante, con el segundo elemento, y así sucesivamente. Esta función debe ser IN PLACE.

3) *Concatena*. Escriba procedimientos para concatenar: a) dos listas L1 y L2 usando `insert`; b) una lista LL de n sublistas usando `insert`; c) una lista LL de n sublistas usando `splice`. Cada procedimiento debe retornar el resultado en una lista nueva.
4) *Invierte*. Escribir una función `void invert(list<int> &L)`, que invierte el orden de la lista `L`. Este algoritmo debe implementarse *in place* y debe ser $O(n)$. Restricción: no utilizar el método `size()`.
5) *Junta*. Escribir una función `void junta(list<int> &L, int c)` que, dada una lista `L`, agrupa de a `c` elementos, dejando su suma *in place*. Por ejemplo, si se le pasa como argumento la lista `L=(1,3,2,4,5,2,2,3,5,7,4,3,2,2)`, después de aplicar el algoritmo `junta(L,3)` debe quedar `L=(6,11,10,14,4)` (notar que se agrupan los últimos elementos, pese a no completar los tres requeridos). El algoritmo debe tener un tiempo de ejecución $O(n)$.
6) *ReemplazaSecuencia*. Dada una lista de enteros `L` y dos listas `SEQ` y `REEMP`, posiblemente de distintas longitudes, escribir una función `void reemplaza(list<int> &L, list<int>& SEQ, list<int> &REEMP)`, que busca todas las secuencias de `SEQ` en `L` y las reemplaza por `REEMP`. Por ejemplo, si `L=(1,2,3,4,5,1,2,3,4,5,1,2,3,4,5)`, `SEQ=(4,5,1)` y `REEMP=(9,7,3)`, entonces después de llamar a `reemplaza(L,SEQ,REEMP)`, debe quedar `L=(1,2,3,9,7,3,2,3,9,7,3,2,3,4,5)`. Para implementar este algoritmo primero buscar desde el principio la secuencia `SEQ`, al encontrarla, reemplazar por `REEMP`, luego seguir buscando a partir del siguiente elemento al último de `REEMP`.
7) *Ascendente1*. Escribir una función `void ascendente1(list<int> &L, list<list<int>> &LL)` que, dada una lista `L`, genera una lista de listas `LL` de tal forma de que cada sublista es ascendente.
8) *Ascendente2*. Escribir una función `void ascendente2(list<int> &L, vector<list<int>> &VL)` que, dada una lista `L`, genera un vector de listas `VL` de tal forma de que cada sublista es ascendente.
9) *Camaleón*. Implemente los predicados `bool menor(int x, int y)`, `bool mayor(int x, int y)` y `bool dist(int x, int y)` que retornen verdadero si x es menor, mayor o menor en valor absoluto que y respectivamente. Luego implemente una función `ordena(list<int> &L, bool (*f)(int,int))` que ordene la lista L dependiendo de la función `f` pasada cómo parámetro.
10) *Problema de Josephus*. Un grupo de soldados se haya rodeado por una fuerza enemiga. No hay esperanzas de victoria si no llegan refuerzos y existe solamente un caballo disponible para el escape. Los soldados se ponen de acuerdo en un pacto para determinar cuál de ellos debe escapar y solicitar ayuda. Forman un círculo y se escoge un número n al azar. Igualmente se escoge el nombre de un soldado. Comenzando por el soldado cuyo nombre se ha seleccionado, comienzan a contar en la dirección del reloj alrededor del círculo. Cuando la cuenta alcanza el valor n, este soldado es retirado del círculo y la cuenta comienza de nuevo, con el soldado siguiente. El proceso continúa de tal manera que cada vez que se llega al valor de n se retira un soldado. El último soldado que queda es el que debe tomar el caballo y escapar. Entonces, dados un número n y una lista de nombres, que es el ordenamiento en el sentido de las agujas del reloj de los soldados en el círculo (comenzando por aquél a partir del cual se inicia la cuenta), escribir un procedimiento `list<string> josephus(list<string>& nombres, int n)` que retorna una lista con los nombres de los soldados en el orden que han de ser eliminados y en último lugar el nombre del soldado que escapa.
11) *Palíndromo*. Escribir un predicado `bool is_palindromo(char* S)`, que reciba como parámetro una cadena de texto `S` y determine si ésta es un palíndromo, ignorando los espacios entre palabras. Un palíndromo es una secuencia de caracteres que se lee igual hacia adelante que hacia atrás, por ejemplo: *alli si maria avisa y asi va a ir a mi silla*. Recordar que un string puede indexarse como un vector. Con el fin de utilizar la estructura `<list>`, primero deben pasarse los elementos del string a una lista y solo utilizar ésta en el algoritmo.
12) *Compacta*. Escribir una función `void compacta(list<int> &L,list<int> &S)` que toma un elemento entero `n` de `S` y, si es positivo, saca `n` elementos de `L` y los reemplaza por su suma. Esto ocurre con todos los elementos de `S` hasta que se acaben, o bien se acaben los elementos de `L`.
13) *maxSubList*. Programar una función `list<int> max_sublist(list<int> &L)` la cual reciba una

lista de enteros y encuentre y retorne la sublista `Lmax` que obtenga la mayor suma entre todos sus elementos. Notar que debido a que algunos elementos pueden ser negativos el problema no se resuelve simplemente tomando todos los elementos. También es posible que la sublista resultado no contenga ningún elemento, en el caso de que todos los elementos de `L` sean negativos. Si hay varias sublistas que den la misma suma, debe retornar la que comience primero y sea más corta. Por ejemplo: `[1,2,-5,4,-3,2]->[4]`, `[5,-3,-5,1,7,-2]->[1,7]`, `[4,-3,11,-2]->[4,-3,11]`.

14) *Merge*. Escribir una función `void merge(list<int> &L1, list<int> &L2, list<int>& L)` la cual recibe dos listas ordenadas (que pueden ser de distinto tamaño) de forma ascendente `L1` y `L2` y retorna una lista `L`, pasada como parámetro, con los elementos de ambas ordenados también en forma ascendente. Por ejemplo si `L1=[1, 3, 6, 11]` y `L2=[2, 4, 6, 10]`, la lista `L` debe quedar como `L=[1, 2, 3, 4, 6, 6, 10, 11]`.

15) *MergeSort*. Programar una función `void mergesort(list<int> &L)` que reciba una lista `L` desordenada y la ordene en forma ascendente mediante la siguiente estrategia recursiva: Si la lista está vacía o tiene un sólo elemento ya está ordenada. Sino se parte la lista en dos sublistas y se las ordena a cada una de forma recursiva. Luego se mezclan (fusionan) cada una de las sublistas ya ordenadas. Para partir una lista puede utilizarse el método de la clase list `void list::splice (iterator position, list& x, iterator first, iterator last)` el cual transfiere los elementos desde la lista `x` al contenedor que realiza la llamada en la posición `position`. Se sugiere utilizar la función `merge(..)` implementada en el punto anterior.

b) **[Pilas y Colas]**

1) *Varios*. Utilizando las operaciones del contenedor `stack<>` de STL, construir una serie de procedimientos que realicen cada una de las actividades siguientes en una pila:
   - a' Asignar `i` al segundo elemento desde la parte superior de la pila, dejando la pila sin sus dos elementos de la parte superior.
   - b' Asignar `i` al segundo elemento desde la parte superior de la pila, sin modificarla
   - c' Dado un entero `n`, asignar `i` al elemento n-ésimo desde la parte superior de la pila, dejando la pila sin sus `n` elementos superiores.
   - d' Asignar `i` al elemento del fondo de la pila, dejando la pila vacía.
   - e' Asignar `i` al elemento del fondo de la pila, dejando la pila sin modificar.
2) *Inverso*. Escribir un procedimiento `bool inverso(list<char>&z)` para determinar si una cadena de caracteres de entrada es de la forma `z=xy` donde `y` es la cadena inversa (o espejo) de la cadena `x`, ignorando los espacios en blanco. Emplear una cola y una pila auxiliares.
3) *Chequeo*. Escribir un segmento de programa que lea expresiones aritméticas del estilo: `((a - b)*(5 - c))/4` y verifique si los paréntesis están embebidos correctamente. Es decir, debemos verificar si: a) Existe igual número de paréntesis a izquierda y derecha; b) Cada paréntesis de la derecha está precedido de su correspondiente paréntesis a la izquierda.
4) *SortStack*. Escribir una función `void sort(list<int> &L)`, que ordena los elementos de `L` de mayor a menor. Para ello emplear el siguiente algoritmo simple, utilizando una pila auxiliar `P`: ir tomando el menor elemento de `L`, eliminarlo de `L` e insertarlo en `P` hasta que `L` este vacía. Luego insertar los elementos de `P` en `L`.
5) *SortQueue*. Escribir una función `void sort(list<int> &L)`, que ordena los elementos de `L` de menor a mayor. Para ello utilizar el siguiente algoritmo simple, utilizando una cola auxiliar `C`: ir tomando el menor elemento de `L`, eliminarlo de `L` e insertarlo en `C` hasta que `L` este vacía. Luego insertar los elementos de `C` en `L`.
6) *PancakeSort*. Dada una pila de números `S`, implementar la función `void pancake_sort(stack<int>&S)` la cual ordena la pila solo haciendo operaciones en las cuales se invierte un rango contiguo de elementos en el tope de la pila.
7) *BurntPancakeSort*. Dada una pila de números `S`, implementar la función `void burnt_pancake_sort(stack<int>&S)` la cual ordena la pila solo haciendo operaciones en las cuales se invierte un rango contiguo de elementos en el tope de la pila teniendo en cuenta de que

cada elemento debe ser invertido un número par de veces (la cara quemada de cada panqueque debe quedar hacia abajo).

8) *Rotación.* Escribir una función `void rotacion(queue <int> &C)`, la cual saca una cierta cantidad de enteros del frente de la cola `C` y los vuelve a insertar en fin de la misma, de tal manera que quede en el frente de cola un número par. Por ejemplo, si `C = [1, 3, 5, 2, 4]` entonces, después de `rotacion(C)`, debe quedar `C = [2, 4, 1, 3, 5]`.

c) [**Correspondencias**]

1) *map2list.* Escribir una función que dado un map `M` retorna las listas de claves y valores, utilice la signatura `void map2list(map<int,int> &M, list<int> &Keys, list<int> &Vals)`. Ejemplo: si `M={1->2, 3->5, 8->20}`, entonces debe retornar `Keys = (1,3,8)` y `Vals = (2,5,20)`.
2) *list2map.* Escribir una función que dadas las listas de claves `(k1,k2,k3, ...)` y valores `(v1,v2,v3, ...)` retorna el map `M` con las asignaciones correspondientes `{ k1->v1, k2->v2, k3->v3, ...}`. Utilice la signatura `void list2map(map<int,int> &M, list<int> &Keys, list<int> &Vals)`. Nota: si hay claves repetidas, sólo debe quedar la asignación correspondiente a la última clave en la lista. Si hay menos valores que claves utilizar cero como valor. Si hay más valores que claves, ignorarlos.
3) *InverseMaps.* Dos correspondencias `M1` y `M2` son inversas una de la otra si tienen el mismo número de asignaciones y para cada par de asignación `x->y` en `M1` existe el par `y->x` en `M2`. Escribir un predicado `bool areinverse(map<int,int> &M1, map<int,int> &M2)`; que determina si las correspondencias `M1` y `M2` son inversas.
4) *mergeMap.* Dadas dos correspondencias `A` y `B`, que asocian enteros con listas ordenadas de enteros, escribir una función `void merge_map(map<int, list<int>> &A, map<int, list<int>> &B, map<int, list<int>> &C)` que devuelve en `C` una correspondencia que asigna al elemento `x` la fusión ordenada de las dos listas `A[x]` y `B[x]`. Si `x` no es clave de `A`, entonces `C[x]` debe ser `B[x]` y viceversa. Sugerencia: utilice la función `merge` implementada en uno de los ejercicios anteriores.
5) *cutoffmap.* Implemente una función `void cutoffmap(map<int, list<int>> &M, int p, int q)` que elimina todas las claves que NO están en el rango `[p,q)`. En las asignaciones que quedan también debe eliminar los elementos de la lista que no están en el rango. Si la lista queda vacía entonces la asignación debe ser eliminada. Por ejemplo: si `M = {1->(2,3,4), 5->(6,7,8), 8->(4,5), 3->(1,3,7)}`, entonces `cutoffmap(M,1,6)` debe dejar `M={1->(2,3,4), 3->(1,3)}`. Notar que la clave `5` ha sido eliminada si bien está dentro del rango porque su lista quedaría vacía. Restricciones: el programa no debe usar contenedores auxiliares.
6) *Aplica.* Escribir una función `void apply_map(list<int> &L, map<int,int> &M, list<int> &ML)` que, dada una lista `L` y una correspondencia `M` retorna por `ML` una lista con los resultados de aplicar `M` a los elementos de `L`. Si algún elemento de `L` no está en el dominio de `M`, entonces el elemento correspondiente de `ML` no es incluido. Por ejemplo, si `L = (1,2,3,4,5,6,7,1,2,3)` y `M= {(1,2),(2,3),(3,4),(4,5),(7,8)}`, entonces, después de hacer `apply_map(L,M,ML)`, debe quedar `ML = {(2,3,4,5,8,2,3,4)}`.
7) *Camino.* Implemente la función `bool es_camino(map<int,list<int>> G, list<int> &L)` que recibe una lista `L` y determina si es o no camino en el grafo `G`. El grafo se representa como un mapa que relaciona cada vértice (clave) con la lista de sus vértices adyacentes (valor).
8) *CompConexa.* Dado un grafo como `map<int,list<int>> G` encontrar los subconjuntos del mismo `list<list<int>> D` que están desconectados, es decir, los conjuntos de vértices de cada una de las componentes conexas. Por ejemplo, si `G={1->{2},2->{1},3->{4},4->{3}}`, entonces debe retornar `D={{1,2},{3,4}}`. La signatura de la función a implementar es `void comp_conexas(map<int,list<int>> &G, list<list<int>> &D)`.
9) *isHamilt.* Dado un grafo `map<int, list<int> >G` y una lista de vértices `list<int> L` determinar si `L` es un camino hamiltoniano en `G`.
10) *Distancia.* Dado un grafo `map<int,list<int>> G` y un vértice de partida `x` se desea determinar la distancia éste al resto de los vértices en `G`. Se solicita retornar una estructura de capas de vecinos de `G` alrededor de `x` definida de la siguiente forma: la capa 0 es `{x}`, la capa 1 son los vecinos de `x`.

A partir de allí la capa $n >= 2$ está formada por los vecinos de los vértices de la capa $n-1$ (es decir la adyacencia de la capa) pero que no están en las capas anteriores (en realidad basta con verificar que no estén en las capas $n-1$ ni $n-2$). Notar que los vértices en la capa n se encuentran a una distancia n del vértices de partida.

11) *ColoreadoGrafos*. Partiendo del modelado de los problemas de coloración de grafos de la Guía 1, implementar funciones que reciban como parámetro el grafo (`map<int,list<int>>`) que modela cada problema y encuentre, mediante una estrategia de coloreo heurística, una solución posible.

12) *Dijkstra*. Programe `float Dijkstra(graphW& G, int u, int v, list<int>&path)` que dado el grafo ponderado `G` definido como `typedef map<int,map<int,float>> graphW;`, implemente el algoritmo de Dijkstra para retornar el costo del camino más corto entre el vértice de partida `u` y el vértice de llegada `v`. Además debe devolver en `path` uno de los posibles caminos. Si no hay camino posible, retornar un número muy grande (infinito).