Part I.

Estructuras de datos

Última modificación: September 16, 2016 13:12.

1. Primer parcial

Pregunta 1: Operaciones y su complejidad. Level M

Considere cada una de las siguientes operaciones y especifique su complejidad. Su respuesta debe estar claramente justificada. Sin embargo, no se requiere la implementación de la operación.

1. Determinar cuántos elementos duplicados existen en una lista doblemente enlazada.
2. Insertar un elemento en una lista ordenada. Es decir, usted puede asumir que antes de producirse la inserción, los elementos dentro de la lista se encuentran en orden. Después de la inserción, la lista debe estar ordenada.
3. Insertar un elemento en un vector ordenado. Asuma que antes de la inserción los elementos del vector están en orden. Después de la inserción el vector debe quedar ordenado.

Pregunta 2: Conceptos de estructuras de datos y sus operaciones. Level M

Responda *falso* o *verdadero* a las siguientes preguntas justificando de forma clara y concisa su respuesta.

1. Insertar un elemento al inicio de una secuencia almacenada en una lista es más fácil (computacionalmente) que si se encuentra almacenada en un vector.
2. Una secuencia va a contener entre 1000 y 1200 elementos. El uso de una lista simplemente enlazada para representar esta secuencia utiliza más memoria que un vector con capacidad inicial 150 y política de crecimiento de 3.

Pregunta 3: Máximo elemento en una pila. Level M

Escriba la operación `max` de la clase `Stack` que debe retornar el máximo elemento insertado hasta el momento en la pila.

```
T& max() { /* Su código aquí */ }
```

Considere el siguiente fragmento como *un posible* uso de la operación:

```
Stack<int> s;
for (int i = 0; i < 6; i++) { s.push(rand()%10); }
s.max() = 15;
```

En el siguiente ejemplo usted puede apreciar el efecto de la operación. La pila de la izquierda es como queda después de ser llenada. La de la derecha es el resultado final.

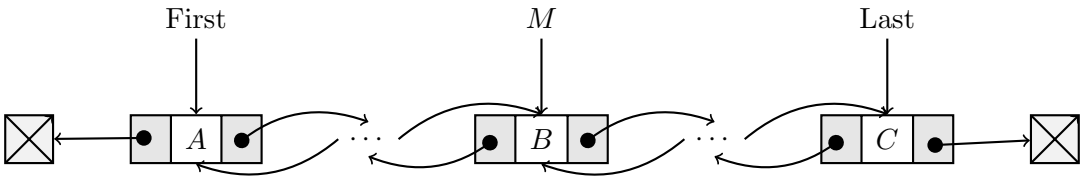
1
5
9
0
1

1
5
15
0
1

Recuerde que una pila puede ser representada de diferentes formas (por ejemplo, utilizando listas, colas, ...). Usted debe especificar claramente que representación utiliza antes de implementar la operación. Es necesario especificar la complejidad de su implementación.

Pregunta 4: Operaciones internas en listas. Level **MH**

La siguiente es una representación de una lista doblemente enlazada. Cada nodo de la lista tiene un dato, un apuntador al nodo anterior y un apuntador al nodo siguiente. En el diagrama $\cdots$ representa la existencia de un número arbitrario de nodos adicionales. M representa un nodo que está dentro de la lista pero que no es ni el primero ni el último. $\boxtimes$ debe ser interpretado como `nullptr`.



Teniendo en cuenta la descripción anterior escriba las líneas de código necesarias para realizar las siguientes operaciones. Puede suponer que los apuntadores ya se encuentran posicionados. Para ninguna de las operaciones usted deberá intercambiar los datos de los nodos. Recuerde especificar la complejidad temporal de cada respuesta.

1. Eliminar todos los nodos a excepción de los que contienen los datos A y C.

2. Eliminar k nodos después del nodo apuntado por M. Si después de M existen menos de k elementos entonces el nodo apuntado por M se convertirá en el último de la lista.

Pregunta 5: Mezcla de listas. Level MH

Como parte de la implementación de un algoritmo de ordenamiento se requiere implementar una operación que mezcle dos listas ordenadas. Por ejemplo, Las listas L y M representan las secuencias $\langle 1, 3, 5, 7, 9\rangle$ y $\langle 2, 4, 6, 8, 10\rangle$. El resultado de `L.sortedMerge(M)` será la lista que representa la secuencia $\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$.
Escriba una implementación eficiente de la operación `sortedMerge` y proponga una cota para su complejidad.

Pregunta 6: Tiempo de ejecución. Level E

Considere las siguientes complejidades de un conjunto de programas: $O\left(n^2\right)$, $O\left(2^n\right)$, $O\left(1\right)$, $O\left(n \cdot \log_2 n\right)$, $O\left(\log_2 n\right)$ y $O\left(n!\right)$. Ordene de mayor a menor las velocidades de los programas ante el mismo valor de n.

Pregunta 7: Operaciones y su complejidad. Level E

Proponga una operación de cualquiera de las estructuras de datos o programas vistos hasta el momento para cada una de las siguientes complejidades.

a) $O\left(n\right)$:

b) $O\left(n^2\right)$:

c) $O\left(n \cdot \log_2 n\right)$:

d) $O\left(\log_2 n\right)$:

e) $O\left(2^n\right)$:

Pregunta 8: Pertenencia en listas. Level E

Escriba una implementación de la operación `member` para la clase `List` desarrollada en clase. La operación verifica si un dato dado se encuentra en la lista. Especifique su complejidad.

Pregunta 9: Conceptos de estructuras de datos y sus operaciones. Level M

Responda *falso* o *verdadero* a las siguientes preguntas justificando clara y de manera concisa su respuesta.

1. Insertar un elemento al inicio de una secuencia almacenada en una lista es más difícil (computacionalmente) que si se encuentra almacenada en un vector.

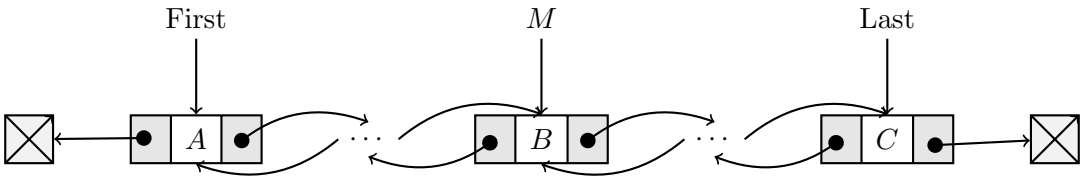
2. Saber si un elemento se encuentra dentro de una secuencia ordenada representada por

una lista doblemente enlazada tiene una complejidad $O(\log_2 n)$ si se usa el algoritmo de búsqueda binaria.

3. Una secuencia va a contener entre 1000 y 1200 elementos. El uso de una lista simplemente enlazada para representar esta secuencia utiliza menos memoria que un vector con capacidad inicial 150 elementos y política de crecimiento de 1.5.

Pregunta 10: Operaciones internas en listas. Level **MH**

La siguiente es una representación de una lista doblemente enlazada. Cada nodo de la lista tiene un dato, un apuntador al nodo anterior y un apuntador al nodo siguiente. En el diagrama "$\cdots$" representa la existencia de un número arbitrario de nodos adicionales. M representa un nodo que está dentro de la lista pero que no es ni el primero ni el último. ☒ debe ser interpretado como `nullptr`.



Teniendo en cuenta la descripción anterior escriba las líneas de código necesarias para realizar las siguientes operaciones. Puede suponer que los apuntadores ya se encuentran posicionados. Para ninguna de las operaciones usted deberá intercambiar los datos de los nodos. Recuerde especificar la complejidad temporal de cada respuesta.

1. Insertar un nodo con dato P inmediatamente después de B.
2. Eliminar el nodo que contiene a C.
3. Intercambiar el nodo con dato M por el nodo con C. Recuerde que no puede intercambiar datos, debe intercambiar los nodos.

Part II.

Programación 3

Pregunta 11: Unificación. Level M

Las dos primeras columnas de la siguiente tabla son expresiones a unificar. Proponga el resultado de la unificación utilizando la tercera columna. En caso de que no se pueda llevar a cabo la unificación describa la razón y coloque `false`.

Expression	Expression	Result	
`[1,2,3]`	`[X|Y]`	`X=`	`Y=`
`[a,[c]]`	`[X|Y]`	`X=`	`Y=`
`[[a]]`	`[X|Y]`	`X=`	`Y=`
`[f(a)]`	`[X|Y]`	`X=`	`Y=`
`[f(a), f(b), c]`	`[X|Y]`	`X=`	`Y=`
`[[a], a]`	`[X|Y]`	`X=`	`Y=`
`[f(Y), 1]`	`[X|Y]`	`X=`	`Y=`
`f([1,2,3])`	`f([X|Y])`	`X=`	`Y=`
`f(X,g(a))`	`g(X,Y)`	`X=`	`Y=`
`f(X,g(a))`	`f(g(Y),Y)`	`X=`	`Y=`
`f(X,g([1,2]))`	`f(f(g(a)),Y)`	`X=`	`Y=`

Pregunta 12: Programa misterio. Level E

Describa que hace el siguiente programa y cuál debe ser la primera respuesta del interprete de Prolog ante la pregunta correspondiente.

```
mistery(X,[X,_]).
mistery(X,[_|L]):- mistery(X,L).
```

```
?- mistery(X,[a,b,c,d]).
```

Pregunta 13: Programa misterio. Level E

Describa que hace el siguiente programa y cuál debe ser la primera respuesta del interprete de Prolog ante la pregunta correspondiente.

```
mistery(L,R) :- otherMistery(L, 0, R).
otherMistery([_],A,A).
otherMistery([H|T],A,R) :-
  A1 is A + H,
  otherMistery(T,A1,R).
```

```
?- mistery([1,2,3,4,5],R).
```

Pregunta 14: Programa misterio. Level E

Describa que hace el siguiente programa y cuál debe ser la primera respuesta del interprete de Prolog ante la pregunta correspondiente.

```
mistery([X|_], 1, X).
mistery([_|Xs], K, Y):-
  K > 1,
  K1 is K-1,
  mistery(Xs,K1,Y).
```

```
?- mistery([a,b,c,d],3,R).
```

Pregunta 15: Número de elementos en lista con listas. Level M

Escriba un predicado llamado `count` que cuente cuántos elementos existen en una lista con un número arbitrario de listas anidadas. Por ejemplo, `?- count([1, [[[1], 2], 3, 4, [[5], [[6]]]],R)` dará como resultado `R=7`.

Pregunta 16: Rotar una lista. Level M

Escriba un predicado llamado `rotate` que rota una lista un número dado de posiciones a la izquierda. Por ejemplo, `?- rotate([a,b,c,d,e,f,g,h],3,R)` dará como resultado `R=[d,e,f,g,h,a,b,c]`.

Part III.

Arquitectura Cliente Servidor

Pregunta 17: Race conditions. Level MH

Una fuente común de problemas en los programas que utilizan concurrencia son los *race conditions*. Considere la siguiente implementación de la clase `Stack` que implementa la estructura de datos conocida como pila.

```
template<typename T>
class Stack {
private:
  std::vector<T> storage;
public:
  Stack() {}
  bool empty() const { return storage.empty(); }
  void push(T elem) { storage.push_back(elem); }
  const T& top() const {
    if(empty()) throw logic_error("Stack is empty");
    return storage.back();
  }
  void pop() {
    if(empty()) throw logic_error("Stack is empty");
    storage.pop_back();
  }
};
```

Adicionalmente considere las siguientes funciones que realizan diferentes cálculos con pilas.

```
void consumer(Stack<int>& s, int e, int& r) {
  r = 0;
  for(int i = 0; i < e; i++) {
    r = r + s.top();
    s.pop();
  }
}
```

```
void consumer2(Stack<int>& s, int& r) {
  r = 0;
  while(!s.empty()) {
    r = r + s.top();
    s.pop();
  }
}
```

```
void producer(Stack<int>& s, int sz, int mx) {
  for(int i = 0; i < sz; i++) { s.push(rand() % mx); }
}
```

1. Considera usted que el siguiente programa está libre de *race conditions*?. Justifique su respuesta. En caso de ser negativa indique los cambios necesarios para corregirlo y describa su salida.

```
int main() {
  Stack<int> s;
  thread p(producer, ref(s), 10, 10));
  p.join();
  int a, b;
  thread t(consumer, ref(s), 5, ref(a));
  thread u(consumer, ref(s), 5, ref(b));
  t.join(); u.join();
  cout << a + b << endl;
  cout << s.size() << endl;
  return 0;
}
```

2. Considera usted que el siguiente programa está libre de *race conditions*?. Justifique su respuesta. En caso de ser negativa indique los cambios necesarios para corregirlo y describa su salida.

```
int main() {
  Stack<int> s;
  thread p(producer, ref(s), 1000, 10));
  p.join();
  thread q(producer, ref(s), 1000, 10));
  q.join();
  thread r(producer, ref(s), 1000, 10));
  thread s(producer, ref(s), 1000, 10));
  r.join(); s.join();
  cout << s.size() << endl;
  return 0;
}
```

3. Considera usted que el siguiente programa está libre de *race conditions*?. Justifique su respuesta. En caso de ser negativa indique los cambios necesarios para corregirlo y describa su salida.

```
int main() {
  Stack<int> s;
  thread p(producer, ref(s), 1, 10));
  int x;
  thread q(consumer2, ref(s), ref(x)));
  p.join(); q.join();
  return 0;
}
```

Pregunta 18: Redes *peer-to-peer*. Level E

Durante la primera parte de nuestro curso tuvimos una presentación muy interesante sobre las redes *peer-to-peer* y en particular *bittorrent*. Responda las siguientes preguntas de acuerdo a lo discutido en dicha presentación.

1. Cuál es el propósito principal de *bittorrent* y cuál es su principal fortaleza que lo diferencia de protocolos centralizados con objetivos similares?.

2. Describa el propósito y la funcionalidad que existe tras la tabla hash distribuida que se utiliza en *bittorrent*.

Pregunta 19: Voz grupal. Level M

Una aplicación de voz sobre *IP* como la que hemos desarrollado durante el curso puede ser extendida de muchas maneras. Una que resulta muy interesante es la de poder tener conversaciones grupales. Es decir, "llamadas" en las que intervienen más de dos personas. Proponga un diseño para adicionar esta funcionalidad a su implementación actual. Tenga presente describir los aspectos más importantes como la interacción entre clientes y servidor, el estado de los componentes del sistema que se deban cambiar y las decisiones de implementación que considere relevantes.

Data structures cheatsheet

Gustavo Gutiérrez
gustavo.ggutierrez@gmail.com

Array

▶ Represents a *tuple* in *set theory*. ▶ Sequential storage in memory. ▶ Fixed size.

Operations

▶Creation: `int a[5];`. `a` stores 5 data of type `int`.
▶Access: `a[3] = 14;`. $O(1)$.

Pros/Cons

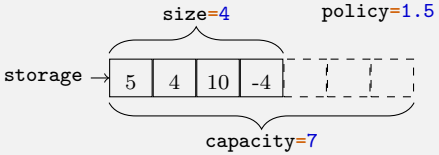
Pros: constant time access to any element by position. *PCons:* need to know the size on creation.

Vector

▶ Dynamic size. ▶ Sequential storage in memory.

Attributes

▶`T* storage`: internal array.
▶`size_t size`: number of inserted elements.
▶`size_t capacity`: maximum number of elements that can be stored in the current array without resizing it.
▶`double policy`: factor to crow the array when resizing.



Operations

▶Creation: ▶ `vector<double> v;` creates an empty vector to store data of type `double`. ▶ `vector<double> w(v);` creates a vector with the same data of `v` O(`v.size`). ▶ `vector<int> x(100);` creates an vector with 100 initial places.
▶Insertion: ▶ `v.push_back(4.2);` inserts 4.2 at the end of `v` in *amortized complexity*. ▶ `v.push_front(1.0);` inserts 1.0 at the first position O(`size`) (shifting all the other elements).
▶Removal: ▶ `v.pop_back();` removes last element $O(1)$. ▶ `v.pop_front();` removes first element O(`size`).
▶Access: `v[5] = 3.14;`. $O(1)$ and only valid positions can be accessed.

Pros/Cons

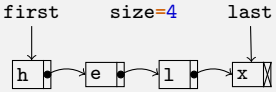
Pros: ▶ Elements can be added as needed. ▶ $O(1)$ access to any element by its position. ▶ Adding or removing elements at the end is $O(1)$ or amortized. *Cons:* ▶ Memory waste on *unsued* places; ▶ adding or removing elements not at the end is O(`size`).

Linked lists

▶ Sequential. ▶ Sparse representation in memory.

Attributes

▶`Node *first`: points to first element.
▶`Node *last`: points to last element.
▶`size_t *size`: stores the number of elements.



Operations

▶Creation: ▶ `list<char> l;` creates an empty list, ▶ `list<char> m(p);` creates a list with the same elements of list `p`.
▶Insertion: ▶ `l.push_back('q');` inserts `'q'` at the end of `l` $O(1)$; while ▶ `l.push_front('x');` inserts `'x'` as the first one $O(1)$.
▶Removal: ▶ `l.pop_front();` removes the firs element of `l` $O(1)$; while ▶ `l.pop_back();` removes the last one O(`size`).
▶Access: ▶ `char w = l.front();` stores in `w` the first element of `l`. ▶ `l.back();` returns the last element.

Pros/Cons

Pros: ▶ Good for insertions and removals in the front and for insertions in the back $O(1)$. *Cons*: ▶ Very bad for searching and removals from the back O(`size`). ▶ Increased memory usage when compared to `vector`.

Stack

▶ Sequential. ▶ *LIFO*: last in, first out.

Attributes

Several options (other sequential DS's.) ▶ `vector`, ▶ `list`, ▶ `dllist`, ▶ `circlist`. Choice influences the complexity of operations.

Operations *(double linked list)*

▶Creation: ▶ `stack<bool> s;` creates empty stack for Booleans $O(1)$. ▶ `stack<bool> t(s);` creates `s` with the same elements of `s` O(`s.size`).
▶Insertion: `s.push(true);` inserts `true`.
▶Removal: `s.pop();` removes the latest inserted element.
▶Access: `bool i = s.top(); i` contains the latest inserted element.
▶Information: ▶ `s.empty();` tests whether `s` is empty. ▶ `s.size();` returns the number of elements.
All operations are $O(1)$.

Queue

▶ Sequential. ▶ *FIFO*: first in, first out.

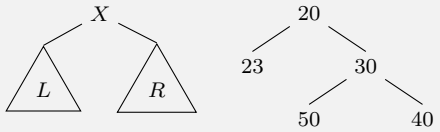
Attributes

Several options (other sequential DS's.) ▶ `vector`, ▶ `list`, ▶ `dllist`, ▶ `circlist`. The choice influences the complexity of operations.

Operations *(linked list)*

▶Creation: ▶ `queue<int> q;` creates empty queue for integers $O(1)$. ▶ `queue<int> t(q);` creates `t` with the same elements of `q` O(`q.size`).
▶Insertion: `q.push(1);` inserts `1`.
▶Removal: `q.pop();` removes the first inserted element.
▶Access: `int i = q.front(); i` contains the first inserted element.
▶Information: ▶ `q.empty();` tests whether `q` is empty. ▶ `q.size();` returns the number of elements.
All operations are $O(1)$.

Binary tree



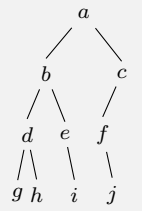
Hierarchical representation of information:
▶Can be empty; or
▶consist of a node with data X with *at most* two *subtrees* L and R called *left* and *right*;
▶both L and R are binary trees themselves.

Terminology

▶ (23) is a *leaf* node as it has no children; ▶ (20) is the *root* of the tree; ▶ (50) and (40) are the *left* and *right* children of (30); ▶ (20) is the *parent* of (23) and (30); ▶ $h = 2$ is the number of arcs in the longest path to a leaf, *height* of the tree.

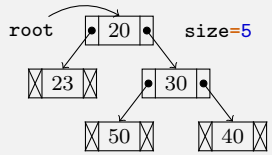
Traversal

Apply a function (*visitor*) to each node of the tree when traversed in certain order.

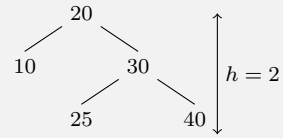


▶Preorder: $\langle a, b, d, g, h, e, i, c, f, j\rangle$
▶Inorder: $\langle g, d, h, b, e, i, a, f, j, c\rangle$
▶Postorder: $\langle g, h, d, i, e, b, j, f, c, a\rangle$
▶Levelorder: $\langle a, b, c, d, e, f, g, h, i, j\rangle$

Implementation



BST (Binary Search Tree)



Binary tree where: ▶ $\forall e \in L : e \prec X$; ▶ $\forall e \in R; X \prec e$; and ▶ both R and L are BSTs.

Operations

▶Creation: `BST<int> t;` for an empty tree.
▶Insertion: `t.insert(10);` inserts 10 in `t`.
▶Information: ▶ `t.empty();` tests wether `t` is empty $O(1)$; ▶ `t.size();` returns the number of elements in `t` $O(1)$; ▶ `t.search(20);` tests whether `t` contains the node (20); ▶ `t.min();` and `t.max();` return the minimum and maximum elements of `t` resp.; ▶ `t.succesor(10);` returns the *smallest* element in `t` that is *greater* than 10 undefined for maximum. ▶ `t.predecessor(10);` returns the *largest* element in `t` that is *smaller* than 10 undefined for minimum.
▶Removal: `t.remove(15);` removes a node with value 15 from `t`.
Unspecified time complexities are $O(h)$.

Pros/Cons

Pros:▶ Highly efficent for search, ▶ insertion and ▶ removal. When the tree is balanced $\log_2$ `size` $\approx h$ *Cons*: ▶ Balance of the tree

depends on the oreder of insertion. Worst case when inserting already sorted elements.

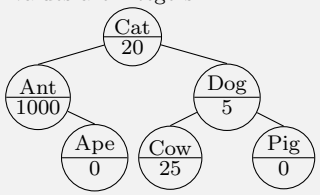
BST (cont.)

Traversal

▶Assume `void f(T& n) { /* ... */};`
▶Assume `using F = function<void(T&)>;`
▶`t.inorder(f);`, `t.preorder(f);`, `t.postorder(f);` and `t.levelorder(f);` apply `f` on each node of `t` in inorder, preorder, postorder and levelorder traversal.
▶`t.interval(T l, T u, F f);` applies `f` on the elements in `t` that belong to the interval $(l, u) = \{x : l \prec x \prec u\}$

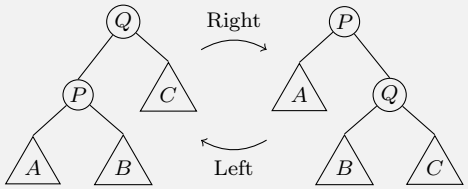
Key-value

▶A BST whose nodes have *two* fields called *key* and *value*.
▶`BST<string,int> t;` creates an empty tree whose nodes keys are strings and their values are integers.



▶`t.insert("Cat",20);` inserts a node with key `"Cat"` and associated value `20`.
▶`t.search("Dog");` returns `{true,5}`. First indicates if the key exists in `t` and if so, second has its value.
▶`t.remove("Ant");` removes a node with key `"Ant"`.

Rotations



▶`rotateRight(q);` (resp. `rotateLeft(p);`) rotates the tree rooted at `q` (resp. `p`) to the right (resp. left).
▶$BST(q) \implies BST($`rotateRight(q)`$)$
▶$BST(p) \implies BST($`rotateLeft(p)`$)$

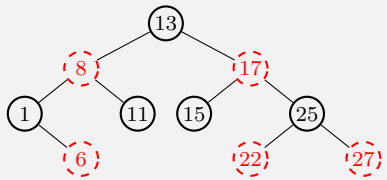
Red black trees

▶ *self* balanced binary search tree. ▶ Tree is kept balanced on insertions and removals. ▶ Its height $h \approx \log_2 n$, n represents its size.

Properties

▶*ordering:* same as in BSTs.
▶*balck height:* the number of black nodes on every path from the root to a each leaf is the same.
▶*coloring:* no two consecutive nodes are red.

Example



Operations

▶Creation: ▶ `RBT<int> r;` creates an empty *red-black tree* to store integers $O(1)$.
▶ `RBT<int> t(r);` creates `t` a copy of `r` $O(n)$.
▶Insertion: `r.insert(10);` inserts `10`
▶Look up: `r.find(5);` tests whether `5` is in the tree.
▶Removal: `r.remove(10);` removes `10`.
`insert`, `find` and `remove` are $O(\log_2 n)$.

Insertion *(detail)*

`t.insert(x);`: insert x using BST insertion and coloring it red. N denotes the current inserted node. Notation: ▶ P: N's parent; ▶ G: N's grand aprent; and ▶ U: N's uncle. Cases after insertion:
▶N is the root node. Paint it black.
▶P is black. Do nothing.
▶P and U are red.P and U are painted black and G becomes red. Fix from G
▶zig-zag conflict with U black. Align N P and G using a right or left rotation.
▶aligned conflict with U black. Use a left or right rotation to balance and exchange the colors of P and G.