INF-134 Estructuras de datos 2024-1

Tarea 2: Gabriel y el super-string

Profesores

Elizabeth Montero
elizabeth.montero@usm.cl

Andrés Navarro
andres.navarrog@usm.cl

Roberto Díaz
roberto.diazu@usm.cl

José Miguel Cazorla
jcazorla@usm.cl

Ricardo Salas
ricardo.salas@usm.cl

Ayudantes Casa Central

Sebastián Torrealba
sebastian.torrealba@usm.cl

Tomás Barros
tomas.barros@sansano.usm.cl

Bastián Jimenez
bjimenez@usm.cl

Franco Cerda
franco.cerda@usm.cl

Ayudantes San Joaquín

Carlos Lagos
carlos.lagosc@usm.cl

Gabriel Escalona
gabriel.escalona@alumnos.inf.utfsm.cl

Juan Alegria
juan.alegria@usm.cl

Lucas Morrison
lucas.morrison@sansano.usm.cl

Giuliana Zanetti
giuliana.zanetti@sansano.usm.cl

Reglas

- La presente tarea debe hacerse en *equipos de dos personas*. Toda excepción a esta regla está sujeta a autorización del ayudante *coordinador* Sebastián Torrealba.
- Debe usarse el lenguaje de programación **C++**. Al realizar la evaluación, las tareas serán compiladas usando el compilador **g++**, usando la línea de comando `g++ archivo.cpp -o output -Wall`. No se aceptarán variantes o implementaciones particulares de **g++**, como el usado por **MinGW** (normalmente ocupado en **Dev C++**). Se deben seguir los tutoriales disponibles en Aula USM.
- No se permite usar la biblioteca STL, así como ninguno de los contenedores y algoritmos definidos en ella (e.g. `vector`, `list`, etc.). Está permitido usar otras funciones de utilidad definidas en bibliotecas estándar, como por ejemplo `math.h`, `string`, `fstream`, `iostream`, etc.
- Recordar que una única tarea en el semestre puede tener nota menor a 30. El incumplimiento de esta regla implica reprobación del curso.

Objetivos de aprendizaje

Entender el funcionamiento de los árboles de búsqueda binaria con modificaciones.

- Aprender cómo funcionan los distintos tipos de recorridos de árboles de búsqueda binaria.
- Comprender el uso de punteros para definir estructuras de tamaño dinámico.
- Entender sobre los posibles usos de los árboles de búsqueda binaria.

Problema a resolver: Gabriel y el super-string

Contexto y problema a resolver

Gabriel es un antiguo profesor de Estructuras de Datos que siente un especial afecto por los strings y se dedica exhaustivamente a estudiarlos. Durante sus estudios se dio cuenta que la implementación de los strings a través de arreglos o listas enlazadas eran poco flexibles y aburridas.

Entonces tuvo una idea, implementar un string con un **árbol de búsqueda binaria**.

Luego de pensar en esa afirmación, Gabriel planteó:

"Puedo hacer un árbol donde la llave del nodo sea el indice del string y el valor del nodo el caracter que representa esa posición"

A ese string le llamo el **super-string**. Luego Garbiel pensó:

"¿Que diferencia significativa tiene esta implementación respecto a las implementaciones clásicas?":

Así que decidió que su implementación tendrá *cinco* operaciones esenciales.

- **Separar:** Esto le permitirá separar la posición i-esima del *super-string* devolviendo dos *super-strings*, uno desde el intervalo $[0, i)$ y otro desde el intervalo $[i, n)$. El segundo *super-string* **puede** estar vacío.
- **Juntar:** Esto le permitirá, dado dos *super-strings*, obtener un nuevo *super-string* concatenando el primero con el segundo.
- **Reverso:** Dado un *super-string* debe revertir los índices. Es decir, si un *super-string* se lee como *"ABCD"*, el reverso del mismo string es *"DCBA"*
- **Recortar:** El **super-string** puede estar representado de una forma que el camino más largo desde la raíz del arbol de busqueda binaria a algún nodo tome una cantidad de $O(n)$ pasos. La función recortar debe permitir pasar esta operación a $O(\log n)$ pasos siempre y cuando no se ha insertado ningún nodo desde ese entonces.

  Un ejemplo de un árbol que toma $O(n)$ pasos para encontrar un nodo se da cuando se intenta crear un árbol de busqueda binaria insertando los siguientes valores en orden:

  $$1, 2, 3, 4, 5 \text{ y } 6$$

  Se asegura que para cualquier lista de nodos existe una disposición donde el camino desde la raíz hasta el nodo más lejano es $O(\log n)$.

  Es importante remarcar que la operación no debe realizar cambios en los nodos, estos deben ser los mismos previos a la operación. Los cambios deben realizarse en términos de la estructura del árbol.
- **stringizar:** Debe retornar un *string* en vez de un *super-string*.

Luego de definir las funciones, Gabriel quiere probarlo en distintos casos de pruebas que permitirán verificar que funciona correctamente. Para esto leerá un archivo llamado `prueba.txt` y hará las siguientes operaciones:

- `REVERSO l r`: Se dan dos números l y r, los cuales "reversan" el orden del substring que esta en $[l, r]$

- `ELIMINAR l r`: Se dan dos números l y r, los cuales "eliminan" el substring que esta en $[l, r]$
- `INSERTAR S i`: Se da un "string" S y se quiere insertar S como "super-string" en la posición i del super-string. Se asegura que el alfabeto posible son las letras del alfabeto inglés en mayúsculas y minúsculas sin contar espacios y con guiones bajos.

  Se asegura que los índices posibles de inserción están entre $[0, n]$, donde n es el tamaño del super-string actual.
- `RECORTAR:` Tiene que recortar el árbol y luego mostrar la altura por pantalla.
- `MOSTRAR:` Se debe mostrar el *super-string* que se lleva en ese momento.
- `FIN:` Se termina la ejecución del programa.

Ejemplo de entrada y salida

Cabe destacar que se probarán distintos casos de prueba. No necesariamente los presentados aquí.

Entrada	**Salida**

`INSERTAR 0 El_Carlos`
`MOSTRAR`
`INSERTAR 9 _es_malo`
`MOSTRAR`
`ELIMINAR 13 16`
`MOSTRAR`
`INSERTAR 13 bueno`
`MOSTRAR`
`RECORTAR`
`REVERSO 0 17`
`MOSTRAR`
`RECORTAR`
`FIN`
	`El_Carlos`
	`El_Carlos_es_malo`
	`El_Carlos_es_`
	`El_Carlos_es_bueno`
	`5`
	`oneub_se_solraC_lE`
	`5`

Entrada	**Salida**

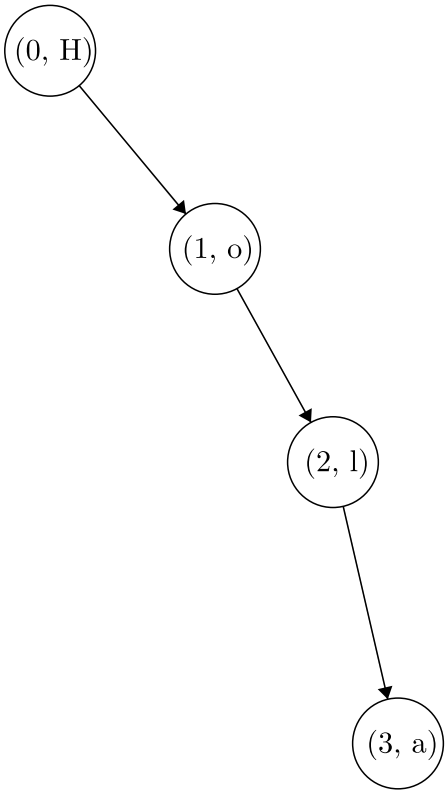
`INSERTAR 0 Gato_`
`MOSTRAR`
`INSERTAR 5 amarillo_`
`MOSTRAR`
`FIN`
	`Gato_`
	`Gato_amarillo_`

Ejemplo de los árboles del super-string

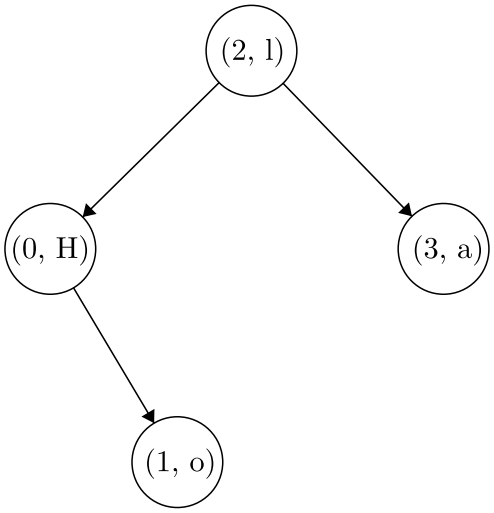
Si se quiere crear un super-string a partir del string “Hola” pueden existir distintos árboles que definen al mismo string. Por simplicidad para mostrar el árbol se define un par de valores, donde el primer elemento del par es el índice del string y el segundo elemento del par es el caracter.

Ambos árboles respetan la propiedad básica de los árboles de búsqueda binaria, donde para cualquier nodo el hijo izquierdo tiene un índice menor y el hijo derecho tiene un índice mayor.

Arbol rama



Arbol recortado



Requisitos al resolver el problema

Se debe usar la siguiente plantilla como **super-string**:

```
class super_string {
  private:
    struct nodo {
      nodo *left = nullptr, *right = nullptr;
      int index;
      char c;
      nodo(int index, char c) {}
    };
    int height = 0; // Altura del árbol
    int length = 0; // Largo del super-string
    nodo* root = nullptr; // Raíz del super-string
  public:
    super_string() {}
    void juntar(super_string &s);
    void agregar(char c); // Insertar un caracter en la última posición
    // En la izquierda esta el super_string a y en la derecha el super_string b
    void separar(int i, super_string &a, super_string &b)
    void reverso(); // No debe cambiar la altura del árbol
    int recortar(); // Retorna this->height después de recortar
    string stringizar(); // Debe ser O(n)
    void limpiar(); // Se deben borrar todos los nodos del super-string
};
```

IMPORTANTE: Se puede agregar *propiedades* o *métodos* a la clase, no asi, modificar lo ya existente, es decir, cambiando parámetros/retornos.

- El metodo llamado `reverso` debe preservar la altura del árbol, es decir, si el árbol tenía altura 5, la aplicación de la funcion **no** debe cambiar dicha altura.
- La complejidad del método `stringizar` debe ser $O(n)$ donde n es la cantidad de nodos.
- El metodo `recortar` apela a la creatividad. Es posible que, comparando su tarea con la de sus pares, encuentren que su funciones recortar obtengan distintas alturas. En este sentido basta con que dichas alturas sean a lo más $O(\log n)$, donde n es la cantidad de nodos. Deben explicar, comentando el código de la función, porqué creen que el método cumple con esa complejidad.

No cumplir con lo solocitado conllevará descuentos en la evaluación de su tarea.

Entrega de la tarea

Entregue la tarea enviando un archivo comprimido zip llamado **tarea1-apellido1-apellido2.zip** (reemplazando sus apellidos según corresponda), en cuyo interior debe estar la tarea dentro de una carpeta también llamada **tarea1-apellido1-apellido2**, a la página *aula.usm.cl* del curso, el cual contenga:

- El **apellido1** es el primer apellido de alguno de los integrantes y el **apellido2** es el primer apellido del otro integrante. Esto significa que el **apellido1** y **apellido2** no son los dos apellidos de alguno de los integrantes.
- Solo **una** persona del grupo debe subir la tarea al aula, no hacer esto, puede conllevar un descuento.
- Los archivos con los códigos fuentes necesarios para el funcionamiento de la tarea. ¡Los archivos deben compilar!
- **nombres.txt**, que debe indicar Nombre, ROL, Paralelo y detalle de qué programó cada integrante del equipo.
- **README.txt**, que debe indicar las instrucciones de compilación en caso de ser necesarias, y la forma de compilación que usó (debe ser alguna de las indicadas en los tutoriales entregados en Aula USM).

Entrega mínima

La entrega mínima permite obtener una nota 30 si cumple con los siguientes requisitos:

- Se crea el *super-string* insertando nodos con agregar.
- Es capaz de mostrar un *super-string* independiente de la complejidad con que se haga.

Restricciones y consideraciones

- Se permite un único día de atraso en la entrega de la tarea. La nota máxima que se puede obtener en caso de entregar con un día de atraso es nota 50.
- Las tareas que no compilen no serán revisadas y tendrán que ser re-corregidas con el ayudante coordinador.
- Debe usar obligatoriamente alguna de las formas de compilación indicadas en los tutoriales entregados en Aula USM.
- Por cada Warning en la compilación se descontarán 5 puntos de la nota.
- Si se detecta COPIA la situación será revisada por ayudante y profesor coordinador.
- La prolijidad, orden y legibilidad del código fuente es obligatoria. Se aplicarán descuentos en la nota si alguno de estos ítems no se cumple.

Directrices de programación

Las directrices corresponden a buenas prácticas de programación, las cuales serán tomadas en consideración para la evaluación de la tarea. El código fuente del programa debe estar estructurado adecuadamente en archivos (separados de ser necesario). Si el código fuente está desordenado, se pueden descontar hasta 20 puntos de la nota. Cada función programada debe tener comentarios de la siguiente forma:

```
/*****
* TipoFunción NombreFunción
```

```
******
* Resumen Función
******
* Input:
* tipoParámetro NombreParámetro : Descripción Parámetro
* .......
******
* Returns:
* TipoRetorno, Descripción retorno
*****/
```

- Por cada comentario faltante, se restarán 5 puntos.
- La indentación (1 TAB o 4 espacios), es muy importante. Por cada bloque mal indentado, se quitarán 10 puntos.