Estructuras de Datos
PRACTICA 3.1

JESUS ANGEL GONZALEZ NOVEZ
GRUPO A 76440070F
PRACTICA 3

Índice

Objetivo de la práctica 3.1

El alumno debe comparar su implementación con los distintos métodos entregados, hacer una crítica de los mismos, enviándome sus opiniones en un fichero pdf. En este caso, no es suficiente con indicar el hecho de que la implementaciones sean diferentes, sino en caso de serlo indicar cuáles han sido las dificultades encontradas a la hora de realizar la práctica.
Dicha crítica se debe entregar el lunes 11 de Noviembre, a las 23:59 horas.

Introducción:

Bueno en mi caso sólo entregué la V1 por tanto solo hablaré sobre esa parte comparándola con la solución. Respecto a la parte V2 pues también le haré una "crítica" pero compararé la V1 corregida con la V2, pues no la implementé. Iré método a método poniendo en la izquierda mi implementación y a la derecha la correcta y comentando mi punto de vista.

VERSION V1

Constructor

diccionario::diccionario(){
entrada e;
e = make_pair("",-1);
diccionario::dic.push_back(e);
}
	diccionario::diccionario(){
	diccionario::entrada nula("",-1);
	dic.push_back(nula);
	pos_max =0;
	}

Ambos declaramos una entrada para la entrada nula pero según entiendo el la inicializa en la declaración y yo le asigno el resultado de make_pair, internamente es lo mismo. Después hacemos un push_back, sin emabargo yo uso el operador de ámbito :: y la solución no lo usa, la cosa esque como entrada la definimos con typedef y tal la debemos llamar con diccionario:: pero dic es un vector, por tanto no hace falta, en el caso entrada usamos :: para heredar todos los métodos del tipo pair en entrada. Y respecto a pos_max , pues se me ha pasado.

Constructor con parámetro

diccionario::diccionario(const entrada & nula){
diccionario::dic.push_back(nula);
}
	diccionario::diccionario(const entrada & nula){
	dic.push_back(nula);
	pos_max =0;
	}

En este caso mas de lo mismo con lo del operador :, pero básicamente hacemos un push_back los dos y respecto a pos_max de nuevo olvidada.

Constructor de copia

diccionario::diccionario (const diccionario & d){
for(size_type i=0; i<d.dic.size(); i++)
diccionario::dic[0] = d.dic[0];
}
	diccionario::diccionario (const diccionario & d){
	dic = d.dic;
	pos_max = d.pos_max;
	}

Aqui nada más ver la solución me dije oh que tonto, olvidé la asignación, la verdad un error tonto, de todas formas un for asignado posición a posición realmente hace lo mismo internamente supongo, respecto a lo del operador :: no lo comentaré más en estos casos pues es la misma explicación que se ha dado anteriormente, y el pos_max una vez mas olvidada!

null()

const diccionario::entrada & diccionario::null()
const {
bool encontrada = false;
size_type i;
for(size_type i=0; i<diccionario::size()
&& !encontrada; i++)
if(diccionario::dic[i].second == -1)
encontrada = true;
return diccionario::dic[i];
}
	const diccionario::entrada & diccionario::null()
	const {
	return dic[0];
	}

Aqui se trataba de devolver la entrada nula, realmente creo más acertada mi solución porque no se si en un futuro iban a cambiar las posiciones de las entradas por alguna razón, por tanto mi lógica me dijo usar una búsqueda secuencial, también cabe decir que podria haberse echo otro tipo de búsquedas claro está, pero no era la idea optimizar la búsqueda, por otro lado decir que si siempre

va a estar en la posición 0, mi bucle for encontrará la entrada nula justo al comenzar, por tanto no hay grandes diferencias, claro está que realiza más instrucciones eso si.

find()

```
const diccionario::entrada &
diccionario::find( const string & s) const{
 bool esta=false;
 size_type i;
 for(i=0; i<diccionario::dic.size()&&!esta; i++)
      if(diccionario::dic[i].first == s)
         esta = true;
      if(esta)
         return diccionario::dic[i];
      else
        return
          diccionario::dic[diccionario::dic.size()-1];
}
```

```
const diccionario::entrada &
diccionario::find( const string & s) const{
  for (int i=1; i< dic.size(); i++)
     if (dic[i].first == s) return dic[i];
  return  null();
}
```

Vayamos por partes, primeramente yo he aplicado la metodología que nos habían enseñado en MP, y eso de meter un return asi en el for y tal lo entiendo ahora, no antes, yo básicamente almaceno la posición de la entrada buscada, detengo el for una vez encontrada y si sé encontró (esta=true) devuelvo dicha posición dic[i] , y en caso contraro devuelvo la última entrada del diccionario, esa parte no se porque la verdad se me iria la olla jaja pues pone bien claro que se retorne null() en caso fallido. Por lo demás hacemos la comparación uno a uno los dos igual y tal por tanto discrepamos en el uso de los returns, sobrentiendo que hacen lo mismo ambas funciones, pero no sabria decir cual tiene más eficiencia.

Operator[]

```
int & diccionario::operator[](const string & s) {

}
```

```
int & diccionario::operator[](const string & s) {
  bool encontrado = false;
  int i;
  for (i=1;i<dic.size() && ! encontrado ;  ){
    if (dic[i].first == s)
      encontrado = true;
    else i++;
  }
  if (!encontrado) {
       dic.push_back(entrada(s,0));
       i = dic.size()-1;
  }
  return dic[i].second;
}
```

Aquí no escribí nada, pero analizando la solución , lo veo bastante claro, primero hace una búsqueda secuencial para ver si está la cadena usando un booleano, si se encontró la devolvemos si no se encontró la insertamos, esto tiene sentido ya que estamos haciendo el operador [] usado para consultar o para insertar.

Operator [] const

const int & diccionario::operator[](const string & s) const{
 	
}
	const int & diccionario::operator[](const string & s) const{
	bool encontrado = false;
	int i;
	for (i=0;i<dic.size() && ! encontrado ;){
	if (dic[i].first == s)
	encontrado = true;
	else i++;
	}
	assert (encontrado == true);
	return dic[i].second;
	}

En este caso tampocó implementé nada, sin embargo no termino de entender su utilidad, básicamente hace lo mismo que la anterior sobrecarga pero da error si no la encuentra, por un lado tengo entendido que el uso de assert en un programa no debe aparecer,es decir se usa solo para hacer debug, por otro lado tendré que consultar con el profesor el porqué de esta doble sobrecarga.

Operator=

diccionario & diccionario::operator=(const diccionario & org){
diccionario::dic.clear();
diccionario::dic = org.dic;
}
	diccionario & diccionario::operator=(const diccionario & org){
	if (this!=&org){
	dic = org.dic;
	pos_max =org.pos_max;
	}
	return *this;
	}

Primero hago un clear, tengo esa costumbre, aunque realmente creo que es inútil, y ahora ya procuraré no usarlo, creí que era conveniente. No hago comprobación de si son iguales con el puntero this, como hace la solución, esta medida evitaría una copia innecesaria si ambos son el mismo. Luego hacemos los dos una asignación normal, y como siempre se me olvida el maldito pos_max !

size()

diccionario::size_type diccionario::size() const {
return diccionario::dic.size();
}
	diccionario::size_type diccionario::size() const {
	return dic.size()-1;
	}

La solución decrementa 1 al valor del size(), sobreentiendo que es debido a que la entrada nula no cuenta, creo que no se ha dicho que esa entrada no cuenta, por ello no le reste 1,pero vaya que de ser asi pues ya esta comprendo y asumo el error y se le pone el -1 en un momento.

empty()

bool diccionario::empty() const{
if(diccionario::dic.size() == 0)
return true;
else
return false;
}
	bool diccionario::empty() const{
	return (dic.size()==1);
	}

Se trataba de comprobar que el size() era 1, es decir solo contiene la entrada nula, yo me equivoqué en que comparaba con 0, lo cual siempre va a ser false, habría que haber puesto 1, pero salvando ese detalle, hacemos los dos lo mismo, lo que pasa que no conocía esa manera de utilizar el return la verdad, pero usándola veo que devuelve true o false de la expresión entre paréntesis, por tanto lo mismo que hago yo pero en una linea!

max_element()

```
const string & diccionario::max_element( )
const{
  int max=0,pos;
  for(unsigned int i=0; i<diccionario::size(); i++)
       if(diccionario::dic[i].second >= max){
          max = diccionario::dic[i].second;
          pos = i;
       }
  return diccionario::[pos].first;
}
```

```
const string & diccionario::max_element( )
const {
  size_type p_max = 0;
  int cuantos = dic[p_max].second;
  for (int i=1; i <  dic.size();i++)
    if (dic[i].second > cuantos) {
      cuantos =dic[i].second;
      p_max = i;
    }
  return dic[p_max].first;
}
```

Bueno en general hemos coincidido en el algoritmo, aunque hay algunas diferencias, la solución empieza en el i=1, es lógico pues la entrada nula es la 0 siempre, cosa que ya hablé arriba, pero bueno, luego vamos comparando y guardando la posición si procede, para finalmente devolver el .first de la pos del dic, igual pero distintos nombres, nada más que añadir creo yo.

cheq_rep()

```
bool diccionario::cheq_rep( ) const {
  if(diccionario::size() == diccionario::dic.size())
          return true;
  else
          return false;
}
```

```
bool diccionario::cheq_rep( ) const {
// Chequearemos puntos i) y iii), pues el 2 no se
puede chequear
  assert( size() == dic.size()-1);
  for (int i=1; i<dic.size(); i++) {
    assert(dic[i].second >=0);
//Valores >= que 0
    for (int j=0; j<dic.size(); j++)
      if (j!=i)
        assert(dic[i].first != dic[j].first);
// No se repite clave, incluido la nula
  }
  return true;
}
```

En esta la verdad esque puse algo por poner pero no lo implementé bien como se puede observar, aparte la comparación que hago la hago mal pues debemos usar dic.size()-1 por lo de la nula y tal, cosa que yo no hago. Analizaré la solución sin más. Primero tenemos un assert lo cual petará el programa en caso de que no se cumpla la condición. Después un for recorremos el dic y abortamos en caso de que algun second excepto el de la nula, sea menor que 0, un segundo recorrido viendo que no hay claves repetidas, y en este caso se incluye a la nula, esto es por que debe evitarse que tengamos dos nulas también. Finalmente devuelve true, esto no quiere decir que siempre devuelva true, si no que en si se devuelve true es que el programa no ha petado, es decir, se han cumplido los assert, de lo contrario no llegaría a devolverse ese true, pararía antes.

Operator<<

ostream & operator<<(ostream & sal, const diccionario & D){
for(unsigned int i=0; i<D.size(); i++)
return (sal << D.dic[i].first << " "
<<D.dic[i].second << endl);
}
	ostream & operator << (ostream & sal, const diccionario & D){
	sal << "size : "<< D.size() <<" => [";
	for (int i=0; i< D.dic.size(); i++)
	sal << "("<< i << ":: " <<D.dic[i].first << ","
	<< D.dic[i].second<<")";
	sal<<"]";
	return sal;
	}

En este caso he hecho return del flujo en cada iteración pensando que estaba bien pero veo que no, nunca me he encontrado con tener que sobrecargar con vectores, siempre lo hemos hecho con cosas sencillas como una fracción o un numero complejo, que no habia que iterar, pero ahora gracias a este ejemplo acabo de comprender bien como se utiliza esto del ostream, a pesar de haber aprobado MP jaja, rellenamos el flujo sal y lo retornamos una vez este todo hecho, claro y apuntado para la próxima.

Hasta aquí la crítica a la V1, ahora compararé la V1 corregida con la V2, pues yo la V2 no la entregué.

VERSION V2

Dado que algunos métodos son idénticos me centraré únicamente en los que difieren de V1 a V2, en la columna izquierda irá V1, y en la derecha V2.

find()

```
const diccionario::entrada &
diccionario::find( const string & s) const{
            for (int i=1; i< dic.size(); i++)
               if (dic[i].first == s) return dic[i];
            return  null();
}
```

```
const diccionario::entrada &
diccionario::find( const string & s) const{
  int inf=1; int sup = dic.size()-1;
  int centro;

  while(inf<=sup){
    centro=(sup+inf)/2;
    if(dic[centro].first==s) return dic[centro];
    else if(s < dic[centro].first ){
       sup=centro-1;
    }
    else {
      inf=centro+1;
    }
  }
  return dic[0];  //Entrada nula
}
```

La V2 usa una búsqueda binaria para encontrar una entrada, esto tiene la ventaja de realizar muchas menos interaciones que la búsqueda secuencial de la V1, pero por otro lado el vector debe estar ordenado para realizarla con éxito, cosa que con la secuencial no ocurre.

Operator[]

```
int & diccionario::operator[](const string & s) {
  bool encontrado = false;
  int i;
  for (i=1;i<dic.size() && ! encontrado ;  ){
    if (dic[i].first == s)
      encontrado = true;
    else i++;
  }
  if (!encontrado) {
       dic.push_back(entrada(s,0));
       i = dic.size()-1;
  }
  return dic[i].second;
}
```

```
int & diccionario::operator[](const string & s) {

  //Paso1 busqueda de s
  int inf=1,  sup = dic.size()-1;
  int centro;
  bool enc=false;
  while(inf<=sup && ! enc){
    centro=(sup+inf)/2;
    if(dic[centro].first==s)
      enc = true;
    else if(s < dic[centro].first ){
       sup=centro-1;
    }
    else {
      inf=centro+1;
    }
  }
  //Paso2 si esta devuelve la ref y si no lo
inserta.
  if (enc) return dic[centro].second;
  else { //No esta:
     diccionario::entrada aux(s,0);
```

	dic.push_back(aux);
	//Insertamos al final, incrementamos el size del vector
	int pos = dic.size()-1;
	while ((pos >1) && (dic[pos-1] > aux)){
	//Desplazamos hasta su posicon
	dic[pos]=dic[pos-1];
	pos--;
	}
	dic[pos]=aux;
	return dic[pos].second;
	}
	}

En este caso se repite lo de la búsqueda binaria para la V2, y luego lo desplaza hasta su posición, en este caso dado el número de instrucciones, debemos ver la eficiencia según el tamaño del diccionario la verdad, porque la V1, realiza menos instrucciones, claro que por otro lado como ya digo dependiendo del tamaño la búsqueda puede ser mejor en un caso u otro.

Operator[] const

const int & diccionario::operator[](const string & s) const{
bool encontrado = false;
int i;
for (i=0;i<dic.size() && ! encontrado ;){
if (dic[i].first == s)
encontrado = true;
else i++;
}
assert (encontrado == true);
return dic[i].second;
}
	const int & diccionario::operator[](const string & s) const{
	int inf=1, sup = dic.size()-1;
	int centro;
	bool enc=false;
	
	while(inf<=sup && ! enc){
	centro=(sup+inf)/2;
	if(dic[centro].first==s)
	enc = true;
	else if(s < dic[centro].first){
	sup=centro-1;
	}
	else {
	inf=centro+1;
	}
	}
	assert(enc==true);
	return dic[centro].second;
	}

Igual que he comentado en el caso anterior, la principal diferencia es el método de búsqueda binario, por lo demás son iguales, los demás detalles están comentados en la crítica del principio de la V1.

cheq_rep()

bool diccionario::cheq_rep( ) const {
// Chequearemos puntos i) y iii), pues el 2 no se puede chequear
 	
assert(size() == dic.size()-1);
	bool diccionario::cheq_rep( ) const {
	// Chequearemos puntos i) y iii), pues el 2 no se puede chequear
	
	assert(size() == dic.size()-1);

```
  for (int i=1; i<dic.size(); i++) {
    assert(dic[i].second >=0);
//Valores >= que 0
    for (int j=0; j<dic.size(); j++)
       if (j!=i)
            assert(dic[i].first != dic[j].first);
// No se repite clave, incluido la nula
  }
  return true;
}
```

```
  for (int i=1; i<dic.size(); i++) {
    assert(dic[i].second >=0);
//Valores >= que 0
    assert(dic[i].first != dic[0].first);
// valores distintos del nulo
    if (i>2)
     assert(dic[i].first != dic[i-1].first);
//Ordenados
  }
  return true;
}
```

En este caso veo más eficiente la solución V2, pues ahorra un for, por lo demás nada más que comentar que no se halla comentado en la crítica de V1.

Aqui finaliza la crítica de ambas versiones.