Universidad de Costa Rica

Facultad de Ingeniería

Escuela de Ciencias de la Computación e Informática

Estructuras de Datos y Análisis de Algoritmos

CI-1221

Grupo 02

PRIMERA TAREA PROGRAMADA

Profesora Sandra Kikut

Elaborado por:

Arnoldo Bustos B51297

Isaac Tretta B47040

Fecha
Jueves 3 de Mayo, 2018

Indice

1 Introducción

. Con este trabajo se busca demostrar los conocimientos adquiridos en el curso de Estructuras de Datos y Análisis de Algoritmos a través de la definicón, implementación y uso de varios modelos matemáticos. Para ello se define cada modelo y sus operadores básicos necesarios para el correcto funcionamiento de las mismas, además de distintos algoritmos que ponen a prueba estos modelos y estructuras de datos, con las que implementaremos los modelos, para demostrar que los resultados teóricos vistos en clase son iguales a los resultados brindados por las pruebas. Estas pruebas se realizaran a través de análisis tiempo-espacio de cada modelo y estructura de datos con diferentes tamaños para analizar correctamente como crecen los tiempos de duración y espacio de una estructura.

2 Objetivos

- Definir, especificar, implementar y usar los modelos Lista Posicionada, Lista Indexada, Lista Ordena y Pila.
- Definir, especificar e implementar algoritmos para los modelos Lista Posicionada, Lista Indexada y Lista Ordenada.
- Usar la Pila como modelo auxiliar para implementar un algoritmo recursivo que no utilice la recursividad provista por el compilador.
- Realizar un análisis de la complejidad computacional de las diferentes estructuras de datos, operadores básicos y algoritmos implementados para cada modelo, tomando en cuenta tiempos reales de ejecución.

3 Enunciado

1. Definir formalmente el modelo Lista Posicionada.

2. Especificar de manera lógica, formal y completa los siguientes operadores básicos de la Lista Posicionada: Iniciar, Destruir, Vaciar, Vacía, Insertar, AgregarAlFinal, Borrar, Recuperar, ModificarElemento, Intercambiar, Primera, Última, Siguiente, Anterior y NumElem. Para cada operador debe incluir: nombre, parámetros con sus tipos y las cláusulas Efecto (claro, completo y conciso), Requiere y Modifica.

3. Definir formalmente el modelo matemático Lista Indexada.

4. Especificar de manera lógica, formal y completa los siguientes operadores básicos de la Lista Indexada: Iniciar, Destruir, Vaciar, Vacía, Insertar, Borrar, Recuperar, ModificarElemento, Intercambiar y NumElem. Para cada operador debe incluir: nombre, parámetros con sus tipos y las cláusulas Efecto (claro, completo y conciso), Requiere y Modifica.

5. Definir formalmente el modelo matemático Lista Ordenada.

6. Especificar de manera lógica, formal y completa los siguientes operadores básicos de la Lista Ordenada: Iniciar, Destruir, Vaciar, Vacía, Agregar, Borrar, Primero, Último, Siguiente, Anterior y NumElem. Para cada operador debe incluir: nombre, parámetros con sus tipos y las cláusulas Efecto (claro, completo y conciso), Requiere y Modifica.

7. Definir formalmente el modelo matemático Pila.

8. Especificar de manera lógica, formal y completa los siguientes operadores básicos de la Pila: Iniciar, Destruir, Vaciar, Vacía, Meter, Sacar, Tope y NumElem. Para cada operador debe incluir: nombre, parámetros con sus tipos y las cláusulas Efecto (claro, completo y conciso), Requiere y Modifica.

4 Desarrollo

4.1 Modelos

4.1.1 Modelo Lista Posicionada

Definición:Se refiere a la sucesión de elementos, no necesariamente ordenada, que tienen entre sí una relación de precedencia. Se tiene la noción de pocisión, la cual es una noción abstracta, que además tiene relación 1:1 con los elementos en la lista. Cuando se realizan agregados se añade campo a la lista y cuando se eliminan elementos de borran campos de la lista.

Operadores Básicos:

- Iniciar(L)
  Efecto: Inicializa la lista L
  Requiere:
  Modifica:

- Destruir(L)
  Efecto: Destruye la lista L
  Requiere: L inicializada
  Modifica:

- Vaciar(L)
  Efecto: Elimina todos los elementos de la lista L
  Requiere: L inicializada
  Modifica: L

- Vacía(L) → Retorna un booleano
  Efecto: Retorna true si la lista L está vacía, retorna falso en caso contrario
  Requiere: L inicializada
  Modifica:

- Insertar(e,p,L)
  Efecto: Inserta el elemento e en la lista L en la posición p
  Requiere: L inicializada, p válida en L
  Modifica: La lista L

- AgregarAlFinal(e,L)
  Efecto: Inserta el elemento e al final de la lista L
  Requiere: L inicializada
  Modifica: L

- Borrar(p,L)
  Efecto: Borra el elemento e en la posición p de la lista L

Requiere: L inicializada
Modifica: L

- Recuperar(p,L) → Retorna un tipo-elemento
  Efecto: Retorna el elemento en la posicion p de la lista L
  Requiere: Lista L inicializada, p válida en L
  Modifica:

- ModificarElemento(e,p,L)
  Efecto: Reemplaza el elemento e en la lista L en la posición p por el que ya existía en dicha posición.
  Requiere: Lista L inicializada, p válida en L
  Modifica: L

- Intercambiar(p_1,p_2,L)
  Efecto: Intercambia de posición a los elementos en las posiciones p_1 y p_2 de la lista L.
  Requiere: L inicializada, p_1 y p_2 válidas en L
  Modifica: L

- Primera(L) → Retorna un tipo-posición
  Efecto: Retorna la primer posición de la lista L.
  Requiere: L inicializada no vacía
  Modifica:

- Última(L) → Retorna un tipo-posicion
  Efecto: Retorna la Última posición de la lista L.
  Requiere: L inicializada no vacía
  Modifica:

- Siguiente(p,L) → Retorna un tipo-posicion
  Efecto: Retorna la posición siguiente de la posición p en la lista L.
  Requiere: L inicializada no vací, p válida en L y p no es la última posición en L
  Modifica:

- Anterior(p,L) → Retorna un tipo-posicion
  Efecto: Retorna la posición anterior de la posición p en la lista L.
  Requiere: L inicializada no vacía, p válida en L y p no es la primera posición en L
  Modifica:

- NumElem(L) → Retorna un integer
  Efecto: Retorna la cantidad de elementos en la lista L.
  Requiere: L inicializada
  Modifica:

4.1.2 Modelo Lista Indexada

Definición: Al igual que el modelo lista posicionada, es una sucesión de elementos con realición de precedencia no necesariamente ordenados con relaciones 1:1 entre las posiciones de la lista y los elementos de la misma, con la diferencia que la "posición" es un índice, es decir un entero.Por ejemplo, si buscamos la posición del tercer elemento de la lista seria el índice 3.

Operadores Básicos:

- Iniciar(L)
  Efecto: Inicializa la lista L
  Requiere:
  Modifica:

- Destruir(L)
  Efecto: Destruye la lista L
  Requiere: L inicializada
  Modifica:

- Vaciar(L)
  Efecto: Elimina todos los elementos de la lista L
  Requiere: L inicializada
  Modifica: L

- Vacía(L) → Retorna un booleano
  Efecto: Retorna true si la lista L está vacía, retorna falso en caso contrario
  Requiere: L inicializada
  Modifica:

- Insertar(e,i,L)
  Efecto: Inserta el elemento e de i-ésimo en la lista L
  Requiere: L inicializada, i ¡= NumElem(L)
  Modifica: L

- Borrar(i,L)
  Efecto: Borra el elemento e i-ésimo de la lista L
  Requiere: L inicializada
  Modifica: L

- Recuperar(i,L) → Retorna un tipo-elemento
  Efecto: Retorna el elemento i-ésimo de la lista L
  Requiere: Lista L inicializada, p válida en L
  Modifica:

- ModificarElemento(e,i,L)
  Efecto: Reemplaza el elemento por el elemento i-ésimo de la lista.
  Requiere: Lista L inicializada, i ¡= NumElem(L)
  Modifica: L

- Intercambiar(i,j,L)
  Efecto: Intercambia de posición a los elementos i-ésimo y j-ésimo de la lista L.
  Requiere: L inicializada, i ¡= NumElem(L), j ¡= NumElem(L)
  Modifica: L

- NumElem(L) → Retorna un integer
  Efecto: Retorna la cantidad de elementos en la lista L.
  Requiere: L inicializada
  Modifica:

4.1.3 Modelo Lista Ordenada

Definición: Es una sucesión de elementos ordenados ascendentemente, a diferencia de los otros modelos no se trabaja por posiciones ni pos índices, solo por los elementos que contiene, lo que genera ambigüedad y tiene el requerimento de no tener ningún elemento repetido.

Operadores Básicos:

- Iniciar(L)
  EFECTO: Inicia la Lista vacía.
  REQUIERE:—-
  MODIFICA:—-

- Destruir(L)
  EFECTO: Destruye la Lista, la deja inutilizable.
  REQUIERE: La Lista inicializada.
  MODIFICA: La Lista.

- Vaciar(L)
  EFECTO: Vacía la Lista pero se puede seguir usando.
  REQUIERE: L inicializada
  MODIFICA: La Lista

- Vacía(L) → Retorna un booleano.
  EFECTO: Retorna true si L esta vacía y false en caso contrario.
  REQUIERE: L inicializada.
  MODIFICA:—-

- Agregar(L, tipo-elemento e)
  EFECTO: Agrega al final de la Lista el elemento e.
  REQUIERE: L inicializado y e no existente en la lista.
  MODIFICA: La Lista.

- Borrar(L, tipo-elemento e)
  EFECTO: Borra el elemento e de la Lista L.
  REQUIERE: L inicializado y e existente en la Lista.
  MODIFICA: La Lista L.

- Primero(L) → Retorna un tipo-elemento e.
  EFECTO: Retorna el primer elemento de la Lista.
  REQUIERE: L inicializado.
  MODIFICA:—-

- Ultimo(L) → Retorna un tipo-elemento e.
  EFECTO: Retorna el ultimo elemento de la Lista.
  REQUIERE: L inicializado.
  MODIFICA:—-

- Siguiente(L, tipo-elemento e) → Retorna un tipo-elemento eSig.
  EFECTO: Retorna el elemento siguiente a e en la Lista.
  REQUIERE: L inicializado y e incluido en la Lista.
  MODIFICA:—-

- Anterior(L, tipo-elemento e) → Retorna un tipo-elemento eSig.
  EFECTO: Retorna el elemento anterior a e en la Lista.
  REQUIERE: L inicializado y e incluido en la Lista.
  MODIFICA:—-

- NumElem(L) → Retorna un integer.
  EFECTO: Retorna el tamaño de la lista.
  REQUIERE: L inicializada.
  MODIFICA:—-

4.1.4 Modelo Pila

Definición: Es un tipo especial de Listas donde los agregados y borrados se hacen por un ”tope”, tambien se les llama Listas ”LIFO”(last in first out) o ”listas ultimo en entrar , primero en salir”. Se pueden visualizar como pilas de libros o de platos donde lo conveniente es agregar y retirarlos desde el tope de los mismos.

Operadores básicos:

- Iniciar(P)
  EFECTO: Inicia la Pila vacía.
  REQUIERE:—-
  MODIFICA:—-

- Destruir(P)
  EFECTO: Destruye la Pila, la deja inutilizable.
  REQUIERE: La Pila inicializada.
  MODIFICA: La Pila.

- Vaciar(P)
  EFECTO: Vacía la Pila pero sigue siendo accesible.
  REQUIERE: P inicializada
  MODIFICA: La Pila.

- Vacío(P) → Retorna un booleano.
  EFECTO: Retorna true si la Pila esta vacía y false en caso contrario.
  REQUIERE: La Pila inicializada.
  MODIFICA:—-

- Meter(P, tipo-elemento e)
  EFECTO: Agrega el elemento e a la Pila.
  REQUIERE: La Pila inicializada.
  MODIFICA: La Pila.

- Sacar(P) → Retorna un tipo-elemento e.
  EFECTO: Borra el elemento e en el tope de la Pila.
  REQUIERE: La Pila inicializada y no vacía.
  MODIFICA: La Pila.

- Tope(P) → Retorna un tipo-elemento e.
  EFECTO: Retorna el elemento en el tope de la Pila para usarlo.
  REQUIERE: La Pila inicializada y no vacía.
  MODIFICA:—-

- NumElem(P) → Retorna un integer.
  EFECTO: Retorna la cantidad de elementos en P.
  REQUIERE: La Pila inicializada.
  MODIFICA:—-

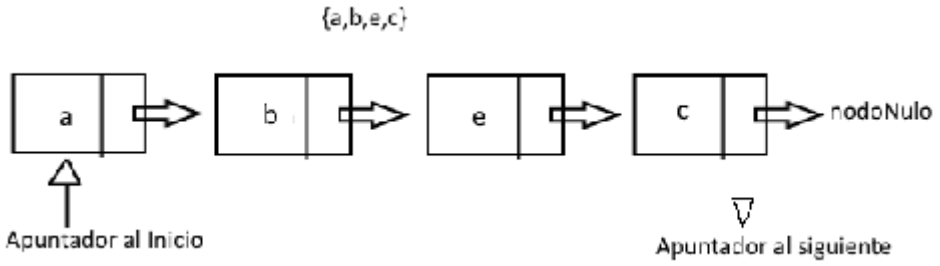
4.2 Estructuras de datos

Se describiran las estructuras de datos usadas para cada modelo, las cuales son:

- Modelo Lista Posicionada: Lista Simplemente Enlazada, Lista Doblemente Enlazada y Arreglo.
- Modelo Lista Indexada: Lista Simplemente Enlazada y Arreglo.
- Modelo Lista Ordenada: Lista Simplemente Enlazada y Arreglo.
- Modelo Pila: Lista Simplemente Enlazada.

4.2.1 Lista Posicionada Simplemente Enlazada

Diagrama y descripción:



Es una estructura de datos donde existe un nodo el cual guarda el elemento y un apuntador al siguiente elemento. Los insertados agregan campo en la lista y los borrados lo eliminan. Los nuevos elementos se pueden insertar en posiciones especificas o al final de la lista sin orden especifico.
Definición en C++ de la estructura de datos:

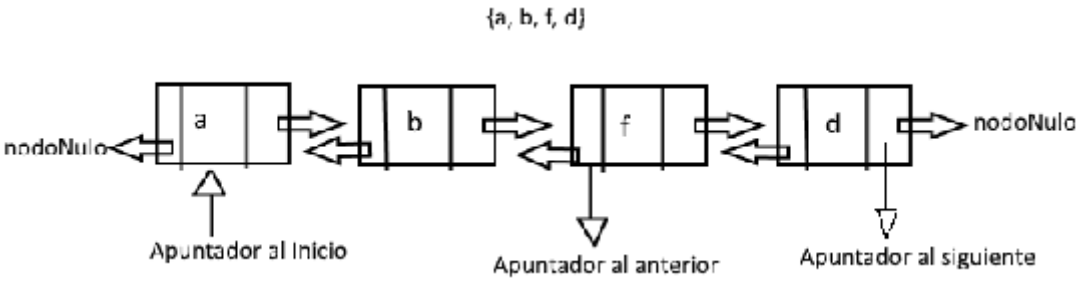
```
template <typename E>
class ListaPosSimEn{

public:
    typedef Nodo<E>* Pos;
    ListaPosSimEn();
    virtual ~ListaPosSimEn();
    void iniciar();
    void destruir();
    void vaciar();
    bool vacia();
    void insertar(E elem, P pos);
    void agregarAlFinal(E elem);
    void borrar(P pos);
    void recuperar(P pos);
    void modificarElemento(E elem,Pos pos);
```

```
17      void   intercambiar(Pos pos1, Pos pos2);
18      Pos primera();
19      Pos ultima();
20      Pos siguiente(Pos pos);
21      Pos anterior(Pos pos);
22      int numElem();
23  private:
24          template < typename Elem>
25          struct Nodo{
26              Elem elemento;
27              Nodo* siguiente;
28              Nodo(): siguiente(nullptr), anterior(nullptr){};
29              Nodo(Elem nuevoElem): elemento(nuevoElem){};
30          };
31          int cantElem;
32          Nodo<E>* inicio;
33          static Nodo<E>* posNula;
34  };
```

4.2.2 Lista Posicionada Doblemente Enlazada

Diagrama y descripción:



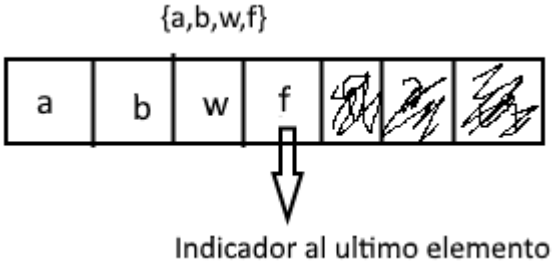
La diferencia con la anterior es la existencia de un puntero al elemento anterior en el nodo. Definición en C++ de la estructura de datos:

```
1   template < typename E>
2   class ListaPosDoEn{
3
4   public:
5       typedef Nodo<E>* Pos;
6       ListaPosDoEn();
7       virtual ~ListaPosDoEn();
8       void iniciar();
9       void destruir();
10      void vaciar();
11      bool vacia();
12      void insertar(E elem, Pos pos);
13      void agregarAlFinal(E elem);
14      void borrar(Pos pos);
15      E recuperar(Pos pos);
16      void modificarElemento(E elem,Pos pos);
17      void   intercambiar(Pos pos1, Pos pos2);
18      Pos primera();
19      Pos ultima();
```

```
    Pos siguiente(Pos pos);
    Pos anterior(Pos pos);
    int numElem();
private:
        template < typename Elem>
        struct Nodo{
            Elem elemento;
            Nodo* siguiente;
            Nodo* anterior;
            Nodo(): siguiente(nullptr), anterior(nullptr){};
            Nodo(E nuevoElem): elemento(nuevoElem){};
        };
        int cantElem;
        Nodo<E>* inicio;
        static Nodo<E>* posNula;
};
```

4.2.3 Lista Posicionada por Arreglo

Diagrama y descripción:



Se trata de un arreglo donde cada casilla guarda el elemento deseado y la posición se entiende como un int representando la casilla deseada.
Definición en C++ de la estructura de datos:

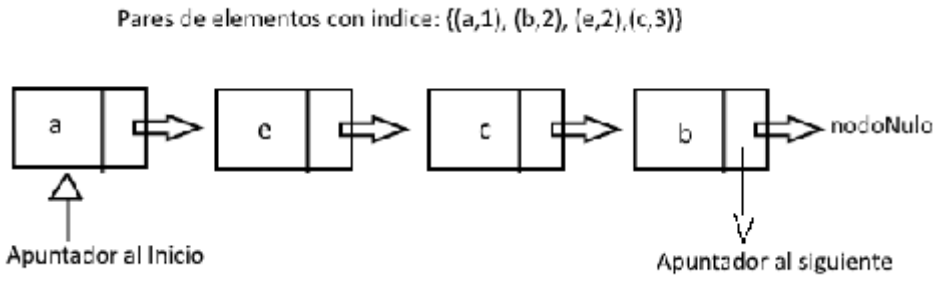
```
#define MAX 100
template < typename E>
class ListaPosArr{

public:
    typedef int Pos;
    ListaPosArr();
    virtual ~listaPosArr();
    void iniciar();
    void destruir();
    void vaciar();
    bool vacia();
    void insertar(E elem, Pos pos);
    void agregarAlFinal(E elem);
    void borrar(Pos pos);
    void recuperar(Pos pos);
    void modificarElemento(E elem,Pos pos);
    void intercambiar(Pos pos1, Pos pos2);
    Pos primera();
    Pos ultima();
```

```
21      Pos siguiente(Pos pos);
22      Pos anterior(Pos pos);
23      int numElem();
24  private:
25          template < typename E>
26          E arreglo[MAX];
27          int cantElem;
28          Pos ultimo;
29          static Pos posNula;
30  };
```

4.2.4 Lista Indexada Simplemente Enlazada

Diagrama y descripción:



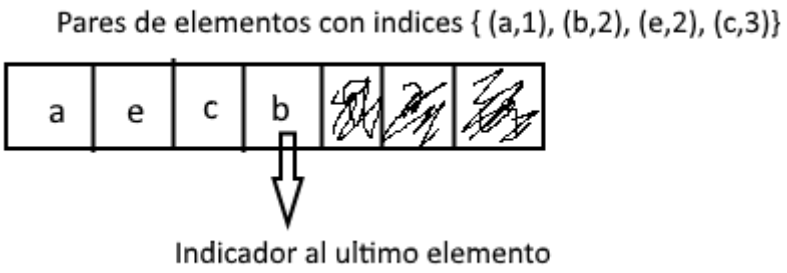
En esta Lista se reciben los elementos a agregar junto a un indice, al insertar y borrar elementos se realizan corrimientos en caso de ser necesario.
Definición en C++ de la estructura de datos:

```
1   template < typename V >
2   class ListaIndEn{
3   public:
4       ListaIndEn();
5       virtual ~ListaIndEn();
6       void iniciar ();
7       void destruir ();
8       void vaciar ();
9       bool vacia ();
10      void insertar (V elemento, int indice);
11      void borrar(int indice);
12      V recuperar(int indice);
13      void modificarElemento(V newE, int indice);
14      void intercambiar(int i, int j);
15      int numElem();
16  private:
17      template < typename T >
18      struct Nodo{
19          T elemento;
20          Nodo *siguiente;
21          Nodo(): siguiente(nullptr){
22          }
23          Nodo(T newE): elemento(newE){
24          }
25      };
```

```
26      int cantidadElem;
27      Nodo<V> *inicio;
28      static Nodo<V> *nodoNulo;
29  };
```

4.2.5 Lista Indexada por Arreglo

Diagrama y descripción:



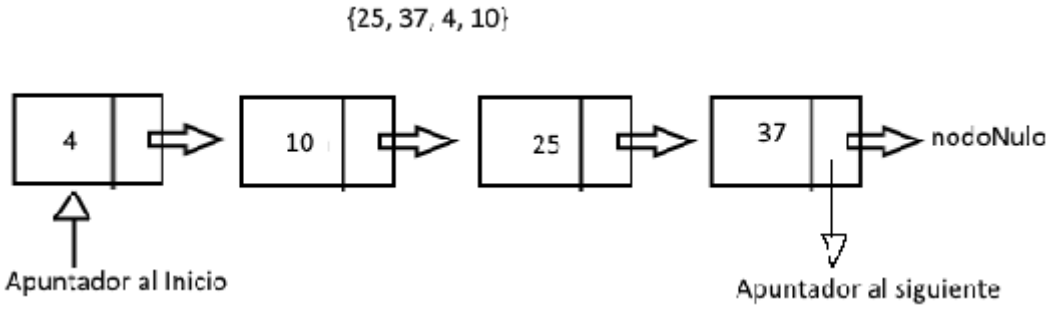
Arreglo donde los agregados y borrados realizan corrimientos completos de los elementos en el mismo. Se tiene una variable donde se guarda el indice del ultimo elemento.
Definición en C++ de la estructura de datos:

```
1   #define MAX 100
2   template < typename E >
3   class ListaIndArr{
4
5   public:
6       ListaIndArr();
7       virtual ~ListaIndArr();
8       void iniciar ();
9       void destruir ();
10      void vaciar ();
11      bool vacia ();
12      void insertar (E elem, int i);
13      void borrar(int i);
14      E recuperar(int i);
15      void modificarElemento(E elem,int i);
16      void intercambiar(int i,int j);
17      int numElem();
18  private:
19      E arreglo[MAX];
20      int cantidadElem;
21      static E elemNulo;
22  };
```

4.2.6 Lista Ordenada Simplemente Enlazada

Diagrama y descripción:

En esta estructura de datos los agregados realizan un movimiento a traves de la lista para verificar donde le corresponde al nuevo elemento estar, ordenando de manera ascedente.

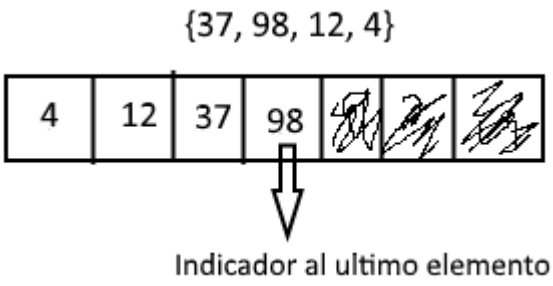


Definición en C++ de la estructura de datos:

```
template < typename V >
class ListaOrdEn{
public:
    ListaOrdEn();
    virtual ~ListaOrdEn();
    void iniciar ();
    void destruir ();
    void vaciar ();
    bool vacia ();
    void agregar(V newE);
    void borrar(V elem);
    V primero();
    V ultimo();
    V siguiente(V elem);
    V anterior(V elem);
    int numElem();
private:
    template < typename T >
    struct Nodo{
        T elemento;
        Nodo *siguiente;
        Nodo(): siguiente(nullptr){
        }
        Nodo(T newE): elemento(newE){
        }
    };
    int cantidadElem;
    Nodo<V> *inicio;
    static Nodo<V> *nodoNulo;
};
```

4.2.7 Lista Ordenada por Arreglo

Diagrama y descripción:

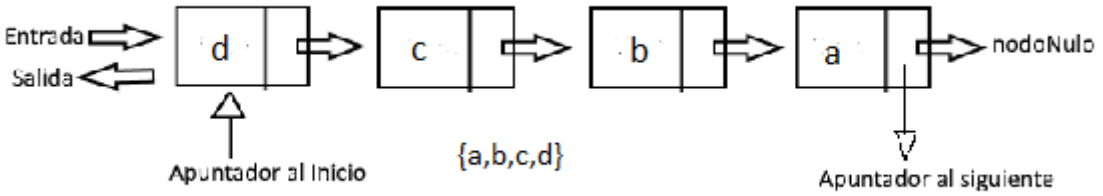


Se trata de un arreglo, que al igual que el anterior, al agregar un elemento se realiza un movimiento a traves del mismo para encontrar donde le corresponde estar. Se realidan corrimientos y además se tiene una variable para guardar el indice del ultimo elemento en el arreglo.
Definición en C++ de la estructura de datos:

```
#define MAX 100
template < typename V >
class ListaOrdArr{
public:
    ListaOrdArr();
    virtual ~ListaOrdArr();
    void iniciar ();
    void destruir ();
    void vaciar ();
    bool vacia ();
    void agregar(V newE);
    void borrar(V elem);
    V primero();
    V ultimo();
    V siguiente(V elem);
    V anterior(V elem);
    int numElem();
private:
    V arreglo[MAX];
    int ultimo;
    int cantidadElem;
    static V elemNulo;
};
```

4.2.8 Pila por Lista Simplemente Enlazada

Diagrama y descripción:



La estructura de datos tiene un apuntador al inicio, cada vez que se realizan agregados este puntero se mueve hacia atras, es decir el nuevo elemento se almacena al inicio de la lista, de esta forma se obtiene el FIFO(First in Last Out).
Definición de C++ de la estructura de datos:

```
template < typename V >
class Pila{
public:
    Pila();
    virtual ~Pila();
    void iniciar();
    void destruir();
    void vaciar();
    bool vacia();
    void meter(V elemento);
    V sacar();
    V tope();
    int numElem();

private:
    template < typename T >
    struct Nodo{
        T elemento;
        Nodo *siguiente;
        Nodo():siguiente(nullptr){
        }
        Nodo(T newE): elemento(newE), siguiente(nullptr){
        }
    };
    int cantidadElem;
    Nodo<V> *top;
    static Nodo<V> *nodoNulo;
};
```

4.3 Algoritmos

4.3.1 Listar

El algoritmo consiste en convertir en una hilera de caracteres los elementos almacenados en la lista.

4.3.2 Simetria

Verifica la simetria en la lista, es decir que sea un palíndromo.

4.3.3 Invertir

Toma la lista he invierte su orden completamente, el primero pasa a ser el ultimo, el segundo el penultimo, y así consecutivamente hasta llegar a la mitad de la lista.

4.3.4 Buscar/Pertene

Verifica la existencia de que tipo-elemento deseado se encuentra en la lista.

4.3.5 Eliminar Elementos Repetidos

Elimina los elementos repetidos en la misma lista, es decir, si la lista es $3, 4, 5, 3, 3, 5$, despúes de aplicar el algorítmo quedaría $3, 4, 5$.

4.3.6 Sublista/Contenida

Se refieren al algoritmo donde se verifica que una Lista esta contenida en su totalidad por otra lista. La diferencia radica que para la sublista no solo debe contener todos los elementos si no además en el mismo orden y sin ningun salto entre los mismos.

4.3.7 Iguales

Es un algoritmo de verificación de igualdad de listas, es decir que ambas tengan los mismos elementos en el mismo orden.

4.3.8 Burbuja Original y Bidireccional

Se trata del algoritmo de ordenamiento por Burbuja. Es similar a selección con la diferencia de en vez de esperar a encontrar al menor de todos los números, cada vez que encuentra uno menor lo intercambia.
En Burbuja Bidireccional al llegar al final de la primera busqueda inicia otra en sentido contrario, asi en el primer ciclo del algoritmo se tiene al elemento menor al principio de la lista y el mayor al final de la misma.

4.3.9 Seleccion

El algoritmo consiste en realizar dos ciclos en la lista, el primero empieza en el primer elemento y el segundo empieza a buscar elementos menores que el primero, cuando encuentra el menor de todos lo guarda en una variable auxiliar y luego intercambia los elementos. Asi continuamente hasta que el primer ciclo llegue al penúltimo elemento de la misma.

4.3.10 Insercion

Se empieza a buscar a travez de la lista haciendo comparaciones de los pares inmediatos, cada vez que se encuentra un par donde el segundo es menor entonces se empieza a buscar donde el elemento debe estar ubicado y se inserta en ese sitio.

4.3.11 Quick Sort

Se trabajaron dos versiones, en una se escoje el pivote del algoritmo por el método de Aho, el cual consiste en recorrer la lista hasta que se encuentre dos números diferentes, y se selecciona el menor para ser el pivote, mientras que en el otro se toma cualquier elemento como pivote y además si la lista es menor a 50 elementos se le aplica el algoritmo de insercion. Luego de conseguir un pivote el algoritmo toma un apuntador al primer elemento y otro al ultimo, se empieza un ciclo para el primer apuntador mientras los elementos sean menores al pivote se mueve el apuntador, al finalizar se hace otro con el ultimo mientras los elementos sean mayores o iguales al pivote se mueve el apuntador hacia atras. Al finalizar ambos ciclos si los apuntadores no se han traslapado se realiza un intercambio de elementos y se vuelven a empezar los ciclos. Cuando se traslapan los apuntadores se realiza una particion de la lista y se empiezan el mismo sistema pero con el ultimo en la posicion de la particion y luego de la particion +1 hasta el final y asi hasta que quede completamente ordenada.

4.3.12 Merge Sort

El Merge Sort consiste en tomar la lista y partirla exactamente a la mitad, luego se llena una lista izquierda con los elementos a la izquierda de la particion y una lista derecha con los elementos a la derecha de la particion. Este proceso se realiza hasta que se tengan listas de un elemento. Una vez se tienen listas de un elemento a travez de llamados recursivos se empieza a devolver a travez de la pila preguntando quien es menor de los elementos e insertandolos en la lista principal ordenamente.

4.3.13 Union

El algoritmo une dos listas, esta union consiste en agregarle los elementos de la segunda lista que no estan en la primera. Asi la primera lista ($Ounatercera$) termina conteniendo los elementos de ambas listas.

4.3.14 Interseccion

Consiste en llenar una tercera lista con los elementos que tienen en común dos listas iniciales.

4.3.15 Diferencia o Eliminar Elementos iguales entre L1 Y L2

Toma la primera Lista y le elimina los elementos iguales contenidos en la segunda lista, dejando en la primera solo los elementos diferentes con relacion a la segunda.

4.3.16 Copiar

El algoritmo toma dos listas y hace que la segunda sea igual a la primera.

5 Manual de Usuario

5.1 Requerimientos de Hardware

El programa no requiere componentes de hardware especificos, por lo que puede ser ejecutado en cualquier arquitectura y configuración. Se elaboro en procesadores Intel de distintas generaciones, todos con 64 bits y sin problemas de ejecucíon.

5.2 Requerimientos de Software

Se requiere del IDE NetBeans 8.2 para el correcto funcionamiento del programa. Se puede descargar del siguiente enlace `https://netbeans.org/download/`

5.3 Arquitectura del programa

El programa principal cuenta con los archivos en la carpeta principal, entre los que se incluye: main.cpp, ListaIndArr.h, ListaIndEn.h, ListaOrdArr.h, ListaOrdEn.h, ListaPosArr.h, ListaPosDoEn.h, ListaPosSimEn.h y Pila.h. Se uso además la herramienta de test de NetBeans, en la cual se crearon un tipo de prueba por modelo y se realizan includes a las estructuras de datos correspondientes.

5.4 Compilación

Para que el proyecto pueda ser compilado con éxito hace falta utilizar dos herramientas, GCC y C++11. GCC es un compilador creado originalmente para ambientes GNU y posteriormente migrado a Windows. Ese compilador permite el uso de C++11, la penúltima entrega del lenguaje, gracias a el, se puede usar declaraciones como nullptr y un mejor uso de la memoria dinámica. Para el archivo make se utilizó la herramienta Msys. Aquí se adjunta el link del sitio principal para descargar el compilador TDM y el Msys en caso de que no lo tenga: `http://tdm-gcc.tdragon.net/index.php/` y `https://goo.gl/pNyTAs`(link acortado).

5.5 Especificación de las funciones del programa

A contuación se le darán una serie de pasos a seguir para lograr que realizar las purebas del proyecto:

- Descomprimir el archivo TP1-CI1221-B47040-B51297.
- Cargar el proyecto en NetBeans.
- Compilar el programa.
- Ejecutar el programa.

- Para probrar las demás clases solo es necesario descomentar los include necesarios y comentar el que se probo previamente. Además de los typedef necesarios. Esta labor se debe realizar tanto en el main como en la clase Algoritmos correspondiente.

6 Datos de prueba

6.1 Formato de los datos de prueba

Se usan Enteros (*int*) en un archivo tipo txt donde se escriben las listas.
Por Ejemplo: El archivo ListaUno.txt es la lista $3, 27, 8, 1, 2, 5, 6$ y en el documento se escriben así (32781256).

6.2 Salida esperada

Se espera que cada algoritmo de su respectivo resultado correcto.

6.3 Salida obtenida

7 Análisis de algoritmos

7.1 Listado y justificación de modelos, operadores y estructuras de datos a analizar

7.2 Casos de estudio, tipos de entrada, tamaños de entrada, diseño de experimentos

7.3 Datos encontrados, tiempos y espacios presentados en tablas y gráficos

7.4 Análisis de los datos

7.5 Comparación de datos reales con los teóricos

7.6 Conclusiones con respecto al análisis realizado

8 Listado de archivos

Los archivos se encuentran en la carpeta principal del proyecto. Se podrian ordenar de la siguiente manera:

1. Pila:
    - Pila.h
2. Lista:
    - ListaIndArr.h
    - ListaIndEn.h
    - ListaOrdArr.h
    - ListaOrdEn.h
    - ListaPosArr.h

- ListaPosDoEn.h
- ListaPosSimEn.h

3. test:
    - testListaInd.cpp
    - testListaOrd.cpp
    - testListaPos.cpp
    - testPila.cpp
4. Algoritmos:
    - AlgListaIndex.h
    - AlgListaOrd.h
    - AlgListaPos.h
    - AlgListaIndex.cpp
    - AlgListaOrd.cpp
    - AlgListaPos.cpp
5. Listas predefinidas:
    - ListaUno.txt
    - ListaDos.txt
    - ListaTres.txt
    - ListaCuatro.txt
6. main.cpp

9 Referencias o bibliografía

- Aho, Alfred y otros. Estructuras de Datos y Algoritmos. Addison-Wesley Iberoamericana. 1988.