ALGORITMOS Y ESTRUCTURA DE DATOS

Operaciones sobre estructuras dinámicas

Antes de comenzar

Este documento resume las principales operaciones que generalmente son utilizadas para la manipulación de las estructuras dinámicas: **lista**, **pila** y **cola**. Cubre además los conceptos de "puntero" y "dirección de memoria" y explica también cómo desarrollar *templates* para facilitar y generalizar el uso de dichas operaciones.

Autor: Ing. Pablo Augusto Sznajdleder.

Revisores: Ing. Analía Mora, Martín Montenegro.

Punteros y direcciones de memoria

Llamamos "puntero" es una variable cuyo tipo de datos le provee la capacidad de contener una dirección de memoria. Veamos:

```
int a = 10;           // declaro la variable a y le asigno el valor 10
int* p = &a;          // declaro la variable p y le asigno "la direccion de a"
cout << *p << endl;   // muestro "el contenido de p"
```

En este fragmento de código asignamos a la variable `p`, cuyo tipo de datos es "puntero a entero", la dirección de memoria de la variable `a`. Luego mostramos por pantalla el valor contenido en el espacio de memoria que está siendo referenciado por `p`; en otras palabras: mostramos "el contenido de `p`".

El operador `&` aplicado a una variable retorna su dirección de memoria. El operador `*` aplicado a un puntero retorna "su contenido".

Asignar y liberar memoria dinámicamente

A través de los comandos `new` y `delete` respectivamente podemos solicitar memoria en cualquier momento de la ejecución del programa y luego liberarla cuando ya no la necesitemos.

En la siguiente línea de código solicitamos memoria dinámicamente para contener un valor entero. La dirección de memoria del espacio obtenido la asignamos a la variable `p`, cuyo tipo es: `int*` (puntero a entero).

```
int* p = new int();
```

Luego, asignamos el valor 10 en el espacio de memoria direccionado por `p`.

```
*p = 10;
```

Ahora mostramos por pantalla el contenido asignado en el espacio de memoria al que `p` hace referencia:

```
cout << *p << endl;
```

Finalmente liberamos la memoria que solicitamos al comienzo.

```
delete p;
```

Nodo

Un nodo es una estructura autoreferenciada que, además de contener los datos propiamente dichos, posee al menos un campo con la dirección de otro nodo del mismo tipo.

Para facilitar la comprensión de los algoritmos que estudiaremos en este documento trabajaremos con nodos que contienen un único valor de tipo `int`; pero al finalizar reescribiremos todas las operaciones como *templates* de forma tal que puedan ser aplicadas a nodos de cualquier tipo.

```
struct Node
{
   int info;  // valor que contiene el nodo
   Node* sig; // puntero al siguiente nodo
};
```

Punteros a estructuras: operador "flecha"

Para manipular punteros a estructuras podemos utilizar el operador `->` (llamado operador "flecha") que simplifica notablemente el acceso a los campos de la estructura referenciada.

Veamos. Dado el puntero `p`, declarado e inicializado como se observa en la siguiente línea de código:

```
Node* p = new Node();
```

Podemos asignar un valor a su campo `info` de la siguiente manera:

```
(*p).info = 10;
```

La línea anterior debe leerse así: "asigno el valor 10 al campo `info` del nodo referenciado por `p`". Sin embargo, el operador "flecha" facilita la notación anterior, así:

```
p->info = 10;
```

Luego, las líneas: `(*p).info = 10` y `p->info = 10` son equivalentes,

Listas enlazadas

Dados los nodos: n_1, n_2,..., n_{i-1}, n_i, n_{i+1},...n_{n-1}, n_n tales que: para todo i>=1 e i<n se verifica que: n_i contiene la dirección de: n_{i+1} y n_n contiene la dirección nula `NULL`, entonces: el conjunto de nodos n_i constituye una lista enlazada. Si `p` es un puntero que contiene la dirección de n_1 (primer nodo) entonces diremos que: "`p` es la lista".

Las operaciones que veremos a continuación nos permitirán manipular los nodos de una lista enlazada.

Agregar un nodo al final de una lista enlazada

La función `add` agrega un nuevo nodo con el valor `x` al final de la lista referenciada por `p`.

```
Node* add(Node*& p, int x)
{
   Node* nuevo = new Node();
   nuevo->info = x;
   nuevo->sig = NULL;
   if( p==NULL )
   {
      p = nuevo;
   }
   else
   {
      Node* aux = p;
      while(aux->sig!=NULL )
      {
         aux = aux->sig;
      }
      aux->sig = nuevo;
   }
   return nuevo;
}
```

Mostrar el contenido de una lista enlazada

La función `mostrar` recorre la lista `p` y muestra por pantalla el valor que contienen cada uno de sus nodos.

```
void mostrar(Node* p)
{
   Node* aux = p;
   while( aux!=NULL )
   {
      cout << aux->info << endl;
      aux = aux->sig;
   }
}
```

Liberar la memoria que ocupan los nodos de una lista enlazada

La función `free` recorre la lista `p` liberando la memoria que ocupan cada uno de sus nodos.

```
void free(Node*& p)
{
   Node* sig;
   while( p!=NULL )
   {
      sig = p->sig;
      delete p;
      p = sig;
   }
}
```

Probar las funciones anteriores

El siguiente programa agrega valores en una lista enlazada; luego muestra su contenido y finalmente libera la memoria utilizada.

```
int main()
{
   // declaro la lista (o, mejor dicho: el puntero al primer nodo
   Node* p = NULL;
   // agrego valores
   add(p,1);
   add(p,2);
   add(p,3);
   add(p,4);
   mostrar(p);
   // libero la memoria utilizada
   free(p);
   return 0;
}
```

Determinar si una lista enlazada contiene o no un valor especificado

La función `find` permite determinar si alguno de los nodos de la lista `p` contiene el valor `v`. Retorna un puntero al nodo que contiene dicho valor o `NULL` si ninguno de los nodos lo contiene.

```
Node* find(Node* p, int v)
{
   Node* aux = p;
   while( aux!=NULL && aux->info!=v )
   {
      aux=aux->sig;
   }
```

```
   return aux;
}
```

Eliminar de la lista al nodo que contiene un determinado valor

La función `remove` permite eliminar de la lista `p` al nodo que contiene el valor `v`.

```
void remove(Node*& p, int v)
{
   Node* aux = p;
   Node* ant = NULL;
   while( aux!=NULL && aux->info!=v )
   {
      ant = aux;
      aux = aux->sig;
   }
   if( ant!=NULL )
   {
      ant->sig = aux->sig;
   }
   else
   {
      p = aux->sig;
   }

   delete aux;
}
```

Eliminar el primer nodo de una lista

La función `removeFirst` elimina el primer nodo de la lista y retorna el valor que este contenía.

```
int removeFirst(Node*& p)
{
   int ret = p->info;
   Node* aux = p->sig;
   delete p;
   p = aux;

   return ret;
}
```

Insertar un nodo al principio de la lista

La función `insertFirst` crea un nuevo nodo cuyo valor será `v` y lo asigna cómo primer ele-mento de la lista direccionada por `p`. Luego de invocar a esta función `p` tendrá la dirección del nuevo nodo.

```
void insertFirst(Node*& p, int v)
{
   Node* nuevo = new Node();
   nuevo->info = v;
   nuevo->sig = p;
   p = nuevo;
}
```

Insertar un nodo manteniendo el orden de la lista

La función `orderedInsert` permite insertar el valor `v` respetando el ctriterio de ordenamiento de la lista `p`; se presume que la lista está ordenada o vacía. Retorna la dirección de memoria del nodo insertado.

```
Node* orderedInsert(Node*& p, int v)
{
   Node* nuevo = new Node();
   nuevo->info = v;
   nuevo->sig = NULL;

   Node* ant = NULL;
   Node* aux = p;

   while( aux!=NULL && aux->info<=v )
   {
      ant = aux;
      aux = aux->sig;
   }

   if( ant==NULL )
   {
      p = nuevo;
   }
   else
   {
      ant->sig = nuevo;
   }

   nuevo->sig = aux;
   return nuevo;
}
```

Ordenar una lista enlazada

La función `sort` ordena la lista direccionada por `p`. La estrategia consiste en eliminar uno a uno los nodos de la lista e insertarlos en orden en una lista nueva; finalmente hacer que `p` apunte a la nueva lista.

```
void sort(Node*& p)
{
   Node* q = NULL;
   while( p!=NULL ){
      int v = removeFirst(p);
      orderedInsert(q,v);
   }

   p = q;
}
```

Insertar en una lista enlazada un valor sin repetición

Busca el valor `v` en la lista `p`. Si no lo encuentra lo inserta respetando el criterio de ordenamiento. Retorna un puntero al nodo encontrado o insertado, y asigna el valor `true` o `false` al parámetro `enc` según corresponda.

```
Node* findAndInsert(Node*& p, int v, bool& enc)
{
   Node* x = find(p,v);
   enc = x!=NULL;

   if( !enc )
   {
      x = orderedInsert(p,v);
   }

   return x;
}
```

Templates

Reprogramaremos todas las funciones que hemos analizado de forma tal que puedan ser utilizadas con listas enlazadas de nodos de cualquier tipo de datos.

Node

```
template <typename T> struct Node
{
   T info;       // valor que contiene el nodo
   Node<T>* sig; // puntero al siguiente nodo
};
```

Función: add

```
template <typename T> Node<T>* add(Node<T>*& p, T x)
{
   // creo un nodo nuevo
   Node<T>* nuevo = new Node<T>();
   nuevo->info = x;
   nuevo->sig = NULL;

   if( p==NULL )
   {
      p = nuevo;
   }
   else
   {
      Node<T>* aux = p;
      while( aux->sig!=NULL )
      {
         aux = aux->sig;
      }

      aux->sig = nuevo;
   }

   return nuevo;
}
```

Función: free

```
template <typename T> void free(Node<T>*& p)
{
   Node<T>* sig;
   while( p!=NULL )
   {
      sig = p->sig;
      delete p;
      p = sig;
   }
}
```

Ejemplo de uso

En el siguiente código creamos dos listas; la primera con valores enteros y la segunda con cadenas de caracteres.

```
int main()
{
   // creo la lista de enteros
   Node<int>* p1 = NULL;
   add<int>(p1,1);
   add<int>(p1,2);
   add<int>(p1,3);
   // la recorro mostrando su contenido
   Node<int>* aux1 = p1;
   while( aux1!=NULL )
   {
      cout << aux1->info << endl;
      aux1 = aux1->sig;
   }
   // libero la memoria
   free<int>(p1);
   // creo la lista de cadenas
   Node<string>* p2 = NULL;
   add<string>(p2,"uno");
   add<string>(p2,"dos");
   add<string>(p2,"tres");
   // la recorro mostrando su contenido
   Node<string>* aux2 = p2;
   while( aux2!=NULL )
   {
      cout << aux2->info << endl;
      aux2 = aux2->sig;
   }
   // libero la memoria
   free<string>(p2);
   return 0;
}
```

Ejemplo con listas de estructuras

```
struct Alumno
{
   int leg;
   string nom;
};
Alumno crearAlumno(int leg, string nom)
{
   Alumno a;
   a.leg = leg;
   a.nom = nom;
   return a;
}
```

```
int main()
{
   Node<Alumno>* p = NULL;
   add<Alumno>(p,crearAlumno(10,"Juan"));
   add<Alumno>(p,crearAlumno(20,"Pedro"));
   add<Alumno>(p,crearAlumno(30,"Pablo"));
```

```
   Node<Alumno>* aux = p;
   while( aux!=NULL )
   {
      cout << aux->info.leg << ", " << aux->info.nom << endl;
      aux = aux->sig;
   }

   free<Alumno>(p);

   return 0;
}
```

Función: `find`

```
template <typename T, typename K>
Node<T>* find(Node<T>* p, K v, int cmpTK(T,K) )
{
   Node<T>* aux = p;
   while( aux!=NULL && cmpTK(aux->info,v)!=0 )
   {
      aux = aux->sig;
   }

   return aux;
}
```

Veamos un ejemplo de como utilizar esta función. Primero las funciones que permiten comparar alumnos

```
int cmpAlumnoLeg(Alumno a,int leg)
{
   return a.leg-leg;
}

int cmpAlumnoNom(Alumno a,string nom)
{
   return strcmp(a.nom.c_str(),nom);
}
```

Ahora analicemos el código de un programa que luego de crear una lista de alumnos le permite al usuario buscar alumnos por legajo y por nombre.

```
int main()
{
   Node<Alumno>* p = NULL;
   add<Alumno>(p,crearAlumno(10,"Juan"));
   add<Alumno>(p,crearAlumno(20,"Pedro"));
   add<Alumno>(p,crearAlumno(30,"Pablo"));

   int leg;
   cout << "Ingrese el legajo de un alumno: ";
   cin >> leg;

   // busco por legajo
   Node<Alumno>* r = find<Alumno,int>(p,leg,cmpAlumnoLeg);

   if( r!=NULL )
   {
      cout << r->info.leg << ", " << r->info.nom << endl;
   }
```

```
   string nom;
   cout << "Ingrese el nombre de un alumno: ";
   cin >> nom;

   // busco por nombre
   r = find<Alumno,string>(p,nom,cmpAlumnoNom);
   if( r!=NULL )
   {
      cout << r->info.leg << ", " << r->info.nom << endl;
   }

   free<Alumno>(p);

   return 0;
}
```

Función: remove

```
template <typename T, typename K>
void remove(Node<T>*& p, K v, int cmpTK(T,K))
{
   Node<T>* aux = p;
   Node<T>* ant = NULL;

   while( aux!=NULL && cmpTK(aux->info,v)!=0 )
   {
      ant = aux;
      aux = aux->sig;
   }

   if( ant!=NULL )
   {
      ant->sig = aux->sig;
   }
   else
   {
      p = aux->sig;
   }

   delete aux;
}
```

Función: removeFirst

```
template <typename T>
T removeFirst(Node<T>*& p)
{
   T ret = p->info;
   Node<T>* aux = p->sig;

   delete p;
   p = aux;

   return ret;
}
```

Función: insertFirst

```
template <typename T>
void insertFirst(Node<T>*& p, T v)
{
   Node<T>* nuevo = new Node<T>();
   nuevo->info = v;
   nuevo->sig = p;
   p = nuevo;
}
```

Función: orderedInsert

```
template <typename T>
Node<T>* orderedInsert(Node<T>*& p, T v, int cmpTT(T,T) )
{
   Node<T>* nuevo = new Node<T>();
   nuevo->info = v;
   nuevo->sig = NULL;

   Node<T>* aux = p;
   Node<T>* ant = NULL;
   while( aux!=NULL && cmpTT(aux->info,v)<=0 )
   {
      ant = aux;
      aux = aux->sig;
   }

   if( ant==NULL )
   {
      p = nuevo;
   }
   else
   {
      ant->sig = nuevo;
   }
   nuevo->sig = aux;

   return nuevo;
}
```

Función: sort

```
template <typename T>
void sort(Node<T>*& p, int cmpTT(T,T))
{
   Node<T>* q = NULL;
   while( p!=NULL )
   {
      T v = removeFirst<T>(p);
      orderedInsert<T>(q,v,cmpTT);
   }

   p = q;
}
```

Función: searchAndInsert

```
template <typename T>
Node<T>* searchAndInsert(Node<T>*& p,T v,bool& enc,int cmpTT(T,T))
{
   Node<T>* x = find<T,T>(p,v,cmpTT);
   enc = x!=NULL;
   if( !enc )
   {
      x = orderedInsert<T>(p,v,cmpTT);
   }
   return x;
}
```

Pilas

Una pila es una estructura restrictiva que admite dos únicas operaciones: *apilar* y *desapilar* o, en inglés: *push* y *pop*. La característica principal de la pila es que el primer elemento que ingresa a la estructura será el último elemento en salir; por esta razón se la denomina LIFO: *Last In First Out*.

Función: push

```
template <typename T> void push(Node<T>*& p, T v)
{
   // se resuelve insertando un nodo al inicio de la lista
   insertFirst<T>(p,v);
}
```

Función: pop

```
template <typename T> T pop(Node<T>*& p)
{
   // se resuelve eliminando el primer nodo de la lista y retornando su valor
   return removeFirst(p);
}
```

Ejemplo de cómo usar una pila

```
int main()
{
   Node<int>* p = NULL;
   push<int>(p,1);
   push <int>(p,2);
   push <int>(p,3);

   while( p!=NULL )
   {
      cout << pop<int>(p) << endl;
   }

   return 0;
}
```

Colas

Una cola es una estructura restrictiva que admite dos únicas operaciones: *enqueue y dequeue*. La característica principal de la cola es que el primer elemento que ingresa a la estructura será también el primero en salir; por esta razón se la denomina FIFO: *First In First Out*.

Podemos implementar una estructura cola sobre una lista enlazada con dos punteros `p` y `q` que hagan referencia al primer y al último nodo respectivamente. Luego, para encolar valor simplemente debemos agregarlo a un nodo y referenciarlo como "el siguiente" de `q`. Y para desencolar siempre debemos tomar el valor que está siendo referenciado por `p`.

Función: `enqueue`

```
template <typename T>
void enqueue(Node<T>*& p, Node<T>*& q, T v)
{
   add<T>(q,v);
   if( p==NULL )
   {
      p = q;
   }
   else
   {
      q = q->sig;
   }
}
```

Función: `dequeue`

```
template <typename T>
T dequeue(Node<T>*& p, Node<T>*& q)
{
   T v = removeFirst<T>(p);

   if( p==NULL )
   {
      q = NULL;
   }
   return v;
}
```

Veamos un ejemplo de como utilizar una cola implementada sobre una lista con dos punteros.

```
int main()
{
   Node<int>* p = NULL;
   Node<int>* q = NULL;
   enqueue<int>(p,q,1);
   enqueue<int>(p,q,2);
   enqueue<int>(p,q,3);
   cout << dequeue<int>(p,q) << endl;
   cout << dequeue<int>(p,q) << endl;
   enqueue<int>(p,q,4);
   enqueue<int>(p,q,5);
   while( p!=NULL )
   {
      cout << dequeue<int>(p,q) << endl;
   }
   return 0;
}
```