Estructuras de datos — Solemne 2

Profesores: Luis Bastías, Rodrigo Paredes, Iván Zuñiga
Ayudantes: Patricia Albornoz, Francisco Claude, Hans Ulloa

Sin apuntes, 1:30 horas

P1. Move-To-Front Lists

Utilizando listas dinámicas simples, implemente el TDA Move-to-Front List. Las operaciones del TDA Move-to-Front List son las siguientes:

a. (2 puntos) **Insert**(x): inserta el elemento x al comienzo de la lista.

b. (2 puntos) **Delete**(x): borra al elemento x de la lista (si es que está).

c. (2 puntos) **Find**(x): busca el elemento x en la lista. Si la búsqueda es exitosa mueve el elemento al comienzo de la lista. Nota: si x es el primer elemento no cambia de posición.

Respuesta

Utilizando listas con un nodo cabeza ficticia *head*. Pseudocodigo a la C++.

---

void **Insert** (Elem x)
1. $head \rightarrow sig$ = new Nodo(x, $head \rightarrow sig$)

Nodo ***Prev** (Elem x)
1. Nodo $*aux = head$
2. **While** $aux \rightarrow sig$!= NULL **Do**
3. **If** $aux \rightarrow sig \rightarrow info == x$ **Then Return** aux // lo encontré
4. **Else** $aux = aux \rightarrow sig$
5. **Return** NULL // x no está en la lista

---

void **Delete** (Elem x)

1. Nodo *$auxP$ = **Prev**(x) // $auxP$ apunta al anterior de x
2. **If** $auxP$ == NULL **Then Return** // x no está en la lista, nada que hacer
3. Nodo *$aux = auxP \rightarrow sig$ // aux apunta al nodo que contiene x
4. $auxP \rightarrow sig = aux \rightarrow sig$ // descuelgo nodo x
5. delete(aux) // libero la memoria del nodo x

---

boolean **Find** (Elem x)

1. Nodo *$auxP$ = **Prev**(x) // $auxP$ apunta al anterior de x
2. **If** $auxP$ == NULL **Then Return** FALSE // x no está en la lista, nada que hacer
3. **If** $auxP == head$ **Then Return** TRUE // x está al comienzo de la lista, fin
4. Nodo *$aux = auxP \rightarrow sig$ // aux apunta al nodo que contiene x
5. $auxP \rightarrow sig = aux \rightarrow sig$
6. $aux \rightarrow sig = head \rightarrow sig$
7. $head \rightarrow sig = aux$
8. **Return** TRUE

Soluciones alternativas de **Delete** y **Find** sin **prev**.

boolean ***Delete** (Elem x)

1. **If** $head \rightarrow sig$ == NULL **Then** Return FALSE // lista vacia, nada que hacer
2. Nodo *$aux = head$
3. **While** $aux \rightarrow sig$!= NULL **Do**
4. **If** $aux \rightarrow sig \rightarrow info == x$ **Then** // lo encontré
5. Nodo *$aux2 = aux \rightarrow sig$ // $aux2$ apunta al nodo que contiene x
6. $aux \rightarrow sig = aux2 \rightarrow sig$ // descuelgo al nodo x
7. delete($aux2$) // libero la memoria del nodo x
8. **Return** TRUE
9. **Else** $aux = aux \rightarrow sig$
10. **Return** FALSE // x no está en la lista

---

boolean **Find** (Elem x)

1. **If** $head \rightarrow sig$ == NULL **Then Return** FALSE // lista vacia, nada que hacer
2. **If** $head \rightarrow sig \rightarrow info == x$ **Then**
3. **Return** TRUE // x está al comienzo de la lista, fin

4. Nodo *$aux = head$
5. **While** $aux \rightarrow sig$!= NULL **Do**
6. **If** $aux \rightarrow sig \rightarrow info == x$ **Then** // lo encontré
7. Nodo *$aux2 = aux \rightarrow sig$ // $aux2$ apunta a x
8. $aux \rightarrow sig = aux2 \rightarrow sig$, $aux2 \rightarrow sig = head \rightarrow sig$, $head \rightarrow sig = aux$
9. **Return** TRUE
10. **Else** $aux = aux \rightarrow sig$
11. **Return** FALSE // x no está en la lista

P2. Colas Dobles

Implemente el TDA Cola Doble (CD) utilizando listas de doble enlace. Las operaciones de la cola doble son:

a. (1 punto) **Insert**(x): inserta x al comienzo de la CD.

b. (1 punto) **Push**(x): inserta x al final de la CD.

c. (1 punto) **Head**(): retorna el valor del primer elemento de la CD.

d. (1 punto) **Tail**(): retorna el valor del ultimo elemento de la CD.

e. (1 punto) **Extract**(): extrae el primer elemento de la CD.

f. (1 punto) **Pop**(): extrae el ultimo elemento de la CD.

Por simpleza, vamos a usar listas doble enlace con *head* y *tail*. Nodos con $\{info, sig, prev\}$.

---

void **Insert** (Elem x)
1. $head \rightarrow sig$ = new Nodo(x, $head \rightarrow sig$, $head$)
2. $head \rightarrow sig \rightarrow sig \rightarrow prev = head \rightarrow sig$

---

void **Push** (Elem x)
1. $tail \rightarrow prev$ = new Nodo(x, $tail$, $tail \rightarrow prev$)
2. $tail \rightarrow prev \rightarrow prev \rightarrow sig = tail \rightarrow prev$

Item **Head** ()

1. **If** $head \rightarrow sig = tail$ **Then Return** NULL// lista vacia, nada que hacer
2. **Else Return** $head \rightarrow sig \rightarrow info$

Item **Tail** ()

1. **If** $head \rightarrow sig = tail$ **Then Return** NULL// lista vacia, nada que hacer
2. **Else Return** $tail \rightarrow prev \rightarrow info$

---

void **Extract** ()

1. **If** $head \rightarrow sig = tail$ **Then Return** // lista vacia, nada que hacer
2. Nodo $*aux = head \rightarrow sig$ // guardo un puntero al primer elemento
3. $head \rightarrow sig = aux \rightarrow sig$
4. $head \rightarrow sig \rightarrow prev = head$
5. delete(aux)

---

void **Pop** ()

1. **If** $head \rightarrow sig = tail$ **Then Return** // lista vacia, nada que hacer
2. Nodo $*aux = tail \rightarrow prev$ // guardo un puntero al ultimo elemento
3. $tail \rightarrow prev = aux \rightarrow prev$
4. $tail \rightarrow prev \rightarrow sig = tail$
5. delete(aux)

P3. Treesort

Considere que tiene una colección de n numeros, con elementos repetidos. Considere que en total son c clases de distintos números. Por ejemplo, en la colección {1 5 2 4 6 1 3 2 3 6 4 2 1 6 3}, se tienen 15 números y hay 6 clases distintas.

a. (4.5 puntos) Proponga un algoritmo que permita ordenar la colección de números utilizando un árbol de búsqueda binaria en tiempo $O(n \log c)$ en promedio.

b. (1.5 puntos) Justifique el costo.

Respuesta

Parte a. Vamos a usar un ABB en que los nodos tienen los campos {Item $info$, int rep, Nodo izq, Nodo der}. Los nodos están ordenados según $info$, que para estos efectos, sin perder generalidad, vamos a suponer que son enteros.

Lo importante acá va a ser modificar el método de inserción en un ABB. Esto es, cuando insertamos un elemento, si es nuevo lo marcamos con una repetición, pero si es antiguo, le incrementamos el contador.

Entonces, primero viene el método de inserción, y luego el que ordena los elementos en con el árbol.

```
ABB *Insert (ABB *raiz, Item x)
      // version recursiva
1.    If raiz == NULL Then raiz = new Nodo(x,1,NULL,NULL)
2.    Else If raiz → info == x Then raiz → rep = raiz → rep + 1
3.    Else If raiz → info > x Then raiz → izq = Insert(raiz → izq, x)
4.    Else raiz → der = Insert(raiz → der, x)
5.    Return raiz
```

```
ABB *Insert (ABB *raiz, Item x)
      // version no recursiva
1.    If raiz = NULL Then Return raiz = new Nodo(x,1,NULL,NULL)
2.    aux = raiz
3.    While aux != NULL Do
4.          If aux → info == x Then aux → rep = aux → rep + 1, Return
5.          Else If aux → info > x Then aux = aux → izq
6.          Else aux = aux → der
7.    aux = new Nodo(x,1,NULL,NULL)
```

```
void Ordena (Lista l)
1.    ABB *a
2.    For i = 0 . . . l.lenght()-1 Do
3.          a = Insert(a, l[i])
```

Parte b. El costo del algoritmo es $O(n \log c)$ en promedio por lo siguiente. Como el ABB tiene c nodos, uno por cada clase, cada inserción tiene costo $O(\log c)$ en promedio. Se realizan n inserciones en este ABB luego el costo total es $O(n \log c)$ en promedio.