Estructuras de Datos Elementales:stacks (pilas), queues (colas), linked lists (listas enlazadas), y rooted trees (árboles con raíz)

Agustín J. González

ELO320: Estructura de Datos y Algoritmos

1 Sem. 2002

Introducción

- Los conjuntos son fundamentales en los computadores como lo son en matemáticas.
- En nuestro campo, los conjuntos cambian (crecen, se reducen, etc). A éstos se les llama ***conjuntos dinámicos***.
- Dependiendo de las operaciones que el algoritmo requiere sobre el conjunto, éstos son conocidos con diferentes nombres.
- Por ejemplo, el conjunto que permite insertar, eliminar, y consultar por pertenencia, es llamado ***diccionario***.
- Operaciones sobre conjuntos dinámicos:
- Search(S,k) retorna $\begin{cases} \text{puntero a un elemento en S tal que su clave es k} \\ \text{nil} \quad \text{Si no hay elementos con clave k} \end{cases}$
- Insert(S,x) Modifica S incorporando el elemento apuntado por x.
- Delete(S,x) Modifica S removiendo el elemento apuntado por x.
- Minimum(S) Retorna el elemento de S con la menor clave.
- Maximum(S) Retorna el elemento de S con la mayor clave.
- Successor(S,x) Retorna el elemento de S cuya clave es la menor no inferior a la clave de x. Si x es el elemento máximo, retorna NIL.
- Predecessor(S,x) Retorna el elemento de S cuya clave es la mayor no superior a la clave de x. Si s es el elemento menor, retorna NIL.

Stacks y Queues

- Los stacks y las queues son conjuntos dinámicos en los que el elemento removido por Delete está predefinido.
- En stacks sólo se puede remover el elemento más recientemente insertado. Implementa así una política **last-in, first-out o LIFO.**
- Las queues sólo pueden remover el elemento más antiguo del conjunto. Implementa una política **first-in, first-out o FIFO**.
- Stacks
  - La operación Insert es llamada aquí PUSH.
  - La operación Delete es llamada POP.
  - Si se hace un pop de un stack vacío, decimos que hay un underflow, lo cual es un error de programa.
  - Si la implementación del stack posee un límite para el número de elementos y éste se excede, decimos que hay un overflow. También es un error.
  - Se incorpora la función TOP que retorna el valor más reciente sin modificar el stack.
- Queues
  - La operación Insert es llamada Enqueue.
  - La operación Delete es llamada Dequeue.
  - Cada queue tiene una head (cabeza) y una tail (cola).
  - También se pueden producir las condiciones de overflow y underflow cuando la implementación tiene capacidad limitada.
  - Se incorpora la función Head que retorna el valor más antiguo de la cola.

Implementación de stack con arreglo

- Const int MAX_ELEMENTS = 100;
- typedef struct stack {
  int top;
  int element [MAX_ELEMENTS];
  } STACK;
- void MakeNull(STACK *S) {
  S->top=-1;
  }
- int Stack_Empty(STACK * S) {
  return (S->top == -1);
  }
- void Push(STACK *S, int x) {
  assert (S->top < MAX_ELEMENTS);
  (*S).top++; /* los paréntesis son requeridos por precedencia de operados */
  (*S).element[(*S).top]=x; /* pudo ser S->element [S->top]=x; */
  /* o ambas en *S.element[++(*S).top]=x; */
  }
- int Pop (STACK *S) {
  assert((*S).top > -1); /* stack no vacío */
  return((*S).element[(*S).top--]);
  }
- int Top(STACK *S) {
  assert(S->top > -1);
  return (S->element[S->top]);
  }

Implementación de queue con arreglo circular

- Const int MAX_ELEMENTS = 100;
- typedef struct stack {
  int head; /* apunta al elemento más antiguo de la queue, el "primero" */
  int tail; /* apunta al elemento más recientemente ingresado a la cola, el de atrás */
  int count; /* cantidad de elemento en la cola. Permite distinguir cola vacía de llena */
  int element [MAX_ELEMENTS];
  } QUEUE;
- void MakeNull(QUEUE *Q) {
  Q->head=0;
  Q->tail=MAX_ELEMENTS-1;
  Q->count=0;
  }
- int Queue_Empty(QUEUE * Q) {
  return (Q->count == 0);
  }

Implementación de queue con arreglo circular (cont)

- Const int MAX_ELEMENTS = 100;
- typedef struct stack {
  int head; /* apunta al elemento más antiguo de la queue, el "primero" */
  int tail; /* apunta al elemento más recientemente ingresado a la cola, el de atrás */
  int count; /* cantidad de elemento en la cola. Permite distinguir cola vacía de llena */
  int element [MAX_ELEMENTS];
  } QUEUE;
- .....
- void Enqueue(QUEUE *Q, int x) {
  assert (Q->count++ < MAX_ELEMENTS);
  Q->tail= ++Q->tail%MAX_ELEMENTS;
  Q->element[Q->tail] = x;
  }
- int Dequeue (QUEUE *Q) {
  int aux=Q->head;
  assert(Q->count-- > 0); /* Queue no vacío */
  Q->head= ++Q->head % MAX_ELEMENTS;
  return((*Q).element[aux]);
  }
- int Head(QUEUE *Q) {
  assert(Q->count >0);
  return (Q->element[Q->head]);
  }

Ejercicio:

Otra forma de distinguir entre cola llena o vacía es dejar libre una casilla al final de la cola. Implemente esta estrategia (elimina la variable count.)

Listas Enlazadas

- Una lista enlazada es una estructura de datos en la que los objetos están ubicados linealmente.
- En lugar de índices de arreglo aquí se emplean punteros para agrupar linealmente los elementos.
- La lista enlazada permite implementar todas las operaciones de un conjunto dinámico.
- En una lista **doblemente enlazada** cada elemento contiene dos punteros (next, prev). Next apunta al elemento sucesor y prev apunta la predecesor.
- Si el predecesor de un elemento es NIL, se trata de la cabeza de la lista.
- Si el sucesor de un elemento es nil, se trata de la cola de la lista.
- Cuando la cabeza es nil, la lista está vacía.
- Una lista puede ser **simplemente enlazada**. En este caso se suprime el campo prev.
- Si la lista está ordenada, el orden lineal de la lista corresponde al orden lineal de las claves.
- En una **lista circular**, el prev de la cabeza apunta a la cola y el next de la cola apunta a la cabeza.

Implementación de Listas doblemente Enlazadas (Sin Centinela)

- typedef struct nodo_lista {
  struct nodo_lista * prev;
  struct nodo_lista * next;
  int elemento;
  } NODO_LISTA;
  typedef NODO_LISTA * P_NODO_LISTA;
- P_NODO_LISTA List_Search(P_NODO_LISTA L, int k) {
  P_NODO_LISTA x = L;
  while (x != NULL) {
  if (x->elemento == k)
  return x;
  x = x->next;
  }
  return (NULL);
  }

Implementación de Listas doblemente Enlazadas (Sin centinela) (cont)

- typedef struct nodo_lista {
  struct nodo_lista * prev;
  struct nodo_lista * next;
  int elemento;
  } NODO_LISTA;
  typedef NODO_LISTA * P_NODO_LISTA;
- /* Inserción sólo al comienzo de la lista */
- void List_Insert (P_NODO_LISTA *pL, P_NODO_LISTA x) {
  x->next = *pL;
  if (*pL != NULL) /* No es lista vacía*/
  (*pL)->prev = x;
  *pL = x;
  x->prev = NULL;
  }
- Si deseamos insertar elementos en cualquier posición, podemos hacerlo usando una lista doblemente enlazada con centinela.

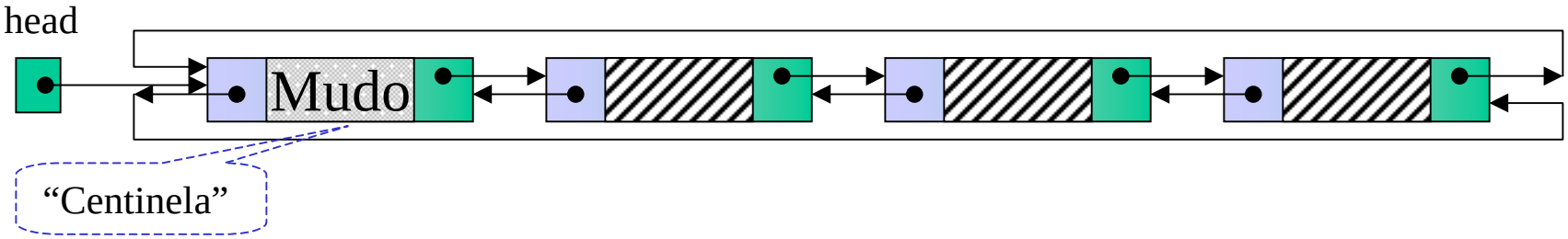
Obs: el detalle de los argumentos formales son parte del protocolo para usar la estructura de datos.
Lo mostrado aquí no es la única opción válida. Los argumentos para List_Insert pudieron ser, por ejemplo, List_Insert(P_NODO_LISTA * posicion, int elemento)
¿Cuál es el tiempo de ejecución de List_Search?
¿Cuál es el tiempo de ejecución de List_Insert?

Implementación de Listas doblemente Enlazadas Sin centinela (Continuación)

- typedef struct nodo_lista {
  struct nodo_lista * prev;
  struct nodo_lista * next;
  int elemento;
  } NODO_LISTA;
  typedef NODO_LISTA * P_NODO_LISTA; /* repetidos aquí por conveniencia para explicar la función de abajo */
- void List_Delete(P_NODO_LISTA * pL, P_NODO_LISTA x) {
  /* esta función asume que x pertenece a la lista. */
  if (x->prev != NULL) /* No se trata del primer elemento */
  x->prev -> next = x->next;
  else
  *pL = (*pL)->next; /* elimina el primer elemento */
  if (x->next != NULL)
  x->next->prev = x->prev;
  }

1.- ¿Cuál es el tiempo de ejecución de List_Delete?
2.- Ejercicios: Implementar las operaciones antes descritas para una lista simplemente enlazada.
3.- Otra opción para implementar listas doblemente enlazadas es hacer unos de un “Centinela”. En este caso la lista es circular y se inicia con un elemento auxiliar o mudo apuntando a si mismo. El código no requiere así tratar en forma especial las condiciones de borde (eliminación del último elemento y eliminación del primero). Hacer código como ejercicio.

Implementación de Listas doblemente Enlazadas Con Centinela



- Void List_Delete’(P_NODO_LISTA x) {
  x->prev->next = x->next;
  x->next->prev = x->prev;
  }
- P_NODO_LISTA List_Search’(P_NODO_LISTA head, int k) {
  P_NODO_LISTA x = head->next;
  while (x != head && x->elemento != k)
  x = x->next;
  return x==head?NULL:x;
  }
- Void List_Insert’ (P_NODO_LISTA pn, P_NODE_LISTA x) {
  /* inserta delante de nodo apuntado por pn*/
  x->next = pn->next;
  pn->next->prev = x;
  pn->next = x;
  x->prev = pn;
  }

Árboles con Raíz

- La idea de usar estructuras de datos enlazados se puede usar también para representar árboles.
- Como el las listas enlazadas supondremos que cada nodo del árbol contiene un campo clave, los restantes campos contienen punteros a otros nodos del árbol.
- Árboles Binarios
- En este caso usamos los campos p, left, y right para almacenar punteros al padre, hijo izquierdo, e hijo derecho de cada nodo del árbol binario.
- Si x.p == NULL, entonces x es la raíz.
- Posible estructura a usar:

```
struct arbol_tag {
        struct arbol_tag p;
        struct arbol_tag left;
        struct arbol_tab right;
};
```

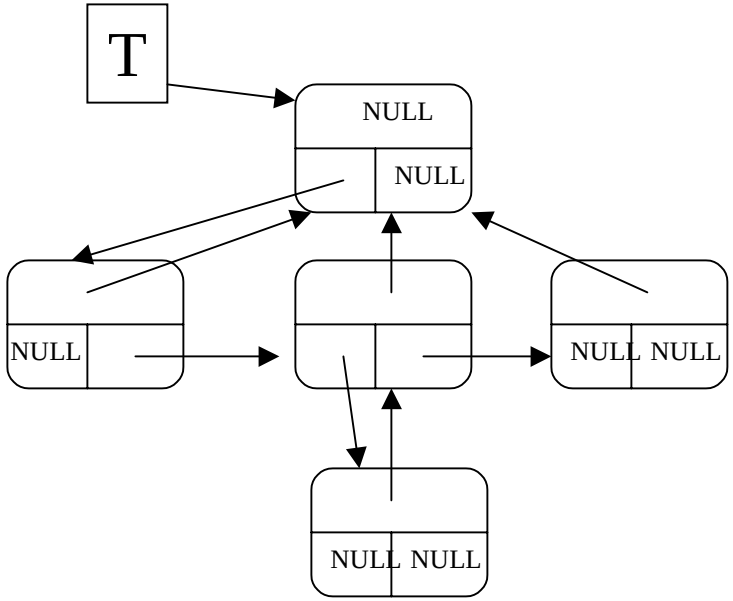
- Para representar árboles con mayor número de hijos se puede agregar nuevos punteros hacia hijo1, hijo2, .., hijon.
- Cuando el número máximo de hijos no está determinado, se puede usar una estructura similar a la de árboles binarios.

Árboles con Raíz (cont)

- En este caso una posible estructura es:

```
struct arbol_tag {
        struct arbol_tag p;  /* apunta hacia el padre */
        struct arbol_tag left; /* hijo de más a la izquierda */
        struct arbol_tab right_sibling; /* hacia el hermano derecho */
};
```

- Si p==NULL => se trata de la raíz
- Si left==NULL => se trata de una hoja.
- Si right_sibling ==NULL => se trata de el hijo de más a la derecha.



Vistazo a Punteros en C

- En C se emplea un * para declarar un puntero. Si p apunta a un carácter, se declara: char *p;
  Una forma de ver esto es usando la interpretación del operador * para desreferenciar. * permite acceder el contenido del valor de la celda apuntada porla variable que acompaña. Ésta debería ser un puntero para que tenga sentido semántico. Es así como la declaración de varias variable de tipo puntero se puede hacer como:
  char *p, *q, *s;
  Esto se puede interpretar como *p es un char, *q es otro char, etc; lo cual indica que p, q, y s son punteros a char.
- Las estructura en C se pueden definir como sigue:
  struct node_tag {
  .....
  };
  La parte de node_tag es opcional pero normalmente es necesaria para referirse nuevamente a la misma estructura cuando definimos o declaramos variables, como en:
  struct node_tag miNodo; /* miNodo es la variable de tipo struct node_tag */
- Para dar claridad a las estructuras de un programa es común que se defina un nuevo tipo y así se le pueda asignar un nombre que represente de mejor forma el tipo de dato. Por ejemplo:
  typedef struct node_tag Node_Type;
  Se crea así un nuevo tipo que no requiere se puede usar como:
  Node_Type miNodo;
- Los campos de una estructura se pueden acceder con el operador **. /* un punto*/**
  Por ejemplo: miNodo.next, cuando next es un campo de la estrucutra Node_Type.
- Cuando se tiene un puntero a una estructura, hay dos formas de acceder a los miembros de ésta.
  Vía operador ->, como en:
  Sea Node_Type *p, miNode;
  ..... /* otras instrucciones, que asignan valor a p */
  miNode.next = p-> next;
  Vía operador *, como en
  miNode.next = (*p).next; /*paréntesis son requeridos por precedencia . sobre * */

Vistazo a Punteros en C (cont)

- Cuando un puntero no tiene definido su valor sus campos no deben ser accedidos para no incurrir en errores de acceso fuera del espacio de memoria del usuario (segmentation fault).

- Un puntero p=NULL; queda mejor definido para explícitamente indicar que su valor no ha sido definido.
  NULL está definido en <stdio.h>

- Para solicitar memoria de la zona de memoria dinámica se emplea el siguiente llamado al sistema:
  Sea Node_Type *p;
  p = (Node_Type *) malloc (sizeof(Node_Type));
  malloc asigna un espacio de memoria de tamaño en bytes dado por el argumento y retorna un puntero a esa zona de memoria.
  sizeof(Node_Type) es un operador interpretado por el compilador y retorna el tamaño en byte requerido por cada variable del tipo indicado.
  (Node_Type *) es necesario para forzar el tipo del valor retornado, de otra manera el compilar no permitiría el acceso a los campos. El compilador diferencia punteros que apuntan a distintos tipos de datos.
  El contenido de la memoria asignada está inicialmente indefinido.

- Para retornar espacio de memoria dinámica no usado por el programa se usa la función ***free***, como en:
  free(p);
  Luego de este llamado el puntero p queda apuntando a un valor indefinido.

Volver