Estructuras autoreferenciadas: listas simples, listas dobles, árboles.

1. Declaración de la estructura.
2. Creación de la estructura.
3. Inserción de un nodo en la estructura.
4. Recorrido de la estructura.
5. Desacoplamiento de la estructura.
6. Eliminación parcial de la estructura.
7. Eliminación total de la estructura.

(1) Declaración.

```c
typedef struct listaSimple {
    int num;
    struct nodo *sigPtr;
} nodo;
```

```c
typedef struct listaDoble {
    struct lista *antPtr;
    int num;
    struct lista *sigPtr;
} nodo;
```

```c
typedef struct arbol {
    struct arbol *izqPtr;
    int num;
    struct arbol *derPtr;
} nodo;
```

(2) Creación.

```c
nodo* crearNodo(void) {
    nodo* nodoPtr;

    nodoPtr = (nodo*)malloc(sizeof(nodo));
    if (nodoPtr == NULL) {
        printf("Memoria insuficiente.\n");
        exit(-1);
    }
    return nodoPtr;
}


    void cargarNodo(nodo *aCargar) {

        printf("Ingrese un entero.\n");
        scanf("%d", &aCargar -> num);

        aCargar -> sigPtr = NULL;
        return;
    }
```

```c
nodo* crearNodo(void) {
    nodo* nodoPtr;

    nodoPtr = (nodo*)malloc(sizeof(nodo));
    if (nodoPtr == NULL) {
        printf("Memoria insuficiente.\n");
        exit(-1);
    }
    return nodoPtr;
}


    void cargarNodo(nodo *aCargar) {

        printf("Ingrese un entero.\n");
        scanf("%d", &aCargar -> num);

        aCargar -> antPtr = NULL;
        aCargar -> sigPtr = NULL;
        return;
    }
```

```c
nodo* crearNodo(void) {
    nodo* nodoPtr;

    nodoPtr = (nodo*)malloc(sizeof(nodo));
    if (nodoPtr == NULL) {
        printf("Memoria insuficiente.\n");
        exit(-1);
    }
    return nodoPtr;
}


    void cargarNodo(nodo *aCargar) {

        printf("Ingrese un entero\n");
        scanf("%d", &aCargar -> num);

        aCargar -> izqPtr = NULL;
        aCargar -> derPtr = NULL;
        return;
    }
```

(3) Inserción.

Pila

```
void insertarAdelante(nodo **cabeza, nodo *aInsertar) {

    aInsertar -> sigPtr = *cabeza;
    *cabeza = aInsertar;
    return;
}
```

Cola

```
void insertarAtras(nodo **cabeza, nodo *aInsertar) {
    nodo *nodoAuxiliar;

    if (*cabeza == NULL) {
        *cabeza = aInsertar;
    } else {
        nodoAuxiliar = *cabeza;
        while ((nodoAuxiliar -> sigPtr) != NULL) {
            nodoAuxiliar = nodoAuxiliar -> sigPtr;
        }
        nodoAuxiliar -> sigPtr = aInsertar;
    }
    return;
}
```

```
void insertarOrdenado(nodo **cabeza, nodo *aInsertar) {
    nodo *nodoActual, *nodoAnterior;

    if (*cabeza == NULL) {
        *cabeza = aInsertar;
    } else {
        nodoActual = *cabeza;
        nodoAnterior = *cabeza;

        while (nodoActual != NULL && (aInsertar -> num > nodoActual -> num)) {
            nodoAnterior = nodoActual;
            nodoActual = nodoActual -> sigPtr;
        }

        if (nodoAnterior == nodoActual) {
            aInsertar -> sigPtr = *cabeza;
            *cabeza = aInsertar;
        } else if (nodoActual != NULL) {
            aInsertar -> sigPtr = nodoActual;
            nodoAnterior -> sigPtr = aInsertar;
        } else {
            nodoAnterior -> sigPtr = aInsertar;
        }
    }
    return;
}
```

Pila

```
void insertarAdelante(nodo **cabeza, nodo **final, nodo *aInsertar) {

    if (*cabeza == NULL) {
        *cabeza = aInsertar;
        *final = aInsertar;
    } else {
        aInsertar -> sigPtr = *cabeza;
        (*cabeza) -> antPtr = aInsertar;
        *cabeza = aInsertar;
    }

    return;
}
```

Cola

```
void insertarAtras(nodo **cabeza, nodo **final, nodo *aInsertar) {

    if (*cabeza == NULL) {
        *cabeza = aInsertar;
        *final = aInsertar;
    } else {
        aInsertar -> antPtr = *final;
        (*final) -> sigPtr = aInsertar;
        *final = aInsertar;
    }

    return;
}
```

```
void insertarOrdenado(nodo **cabeza, nodo **final, nodo *aInsertar) {
    nodo *nodoActual, *nodoAnterior;

    if (*cabeza == NULL) {
        *cabeza = aInsertar;
        *final = aInsertar;
    } else {
        nodoActual = *cabeza;
        nodoAnterior = *cabeza;
        while (nodoActual != NULL && (aInsertar -> num > nodoActual -> num)) {
            nodoAnterior = nodoActual;
            nodoActual = nodoActual -> sigPtr;
        }
        if (nodoAnterior == nodoActual) {
            aInsertar -> sigPtr = *cabeza;
            (*cabeza) -> antPtr = aInsertar;
            *cabeza = aInsertar;
        } else if (nodoActual != NULL) {
            aInsertar -> sigPtr = nodoActual;
            aInsertar -> sigPtr = nodoAnterior;
            nodoActual -> antPtr = aInsertar;
            nodoAnterior -> sigPtr = aInsertar;
        } else {
            aInsertar -> antPtr = *final;
            (*final) -> sigPtr = aInsertar;
            *final = aInsertar;
        }
    }

    return;
}
```

```
void insertarNodo(nodo **raiz, nodo *aInsertar) {

    if (*raiz == NULL) {
        *raiz = aInsertar;
    } else {
        if (aInsertar -> num < (*raiz) -> num) {
            insertarNodo(&(*raiz)->izqPtr, aInsertar);
        } else if (aInsertar -> num > (*raiz) -> num) {
            insertarNodo(&(*raiz)->derPtr, aInsertar);
        } else {
            printf("El num se encuentra duplicado.\n");
        }
    }

    return;
}
```

(4) Recorridos.

```c
void mostrarLista(nodo *cabeza) {

    if (cabeza == NULL) {
        printf("La lista esta vacia.\n");
    } else {
        while (cabeza != NULL) {
            printf("%d\n", cabeza -> num);
            cabeza = cabeza -> sigPtr;
        }
    }
    return;
}
```

```c
void mostrarLista(nodo *cabeza) {

    if (cabeza == NULL) {
        printf("La lista esta vacia.\n");
    } else {
        while (cabeza != NULL) {
            printf("%d\n", cabeza -> num);
            cabeza = cabeza -> sigPtr;
        }
    }
    return;
}


void mostrarInvertida(nodo *final) {

    if (final == NULL) {
        printf("La lista esta vacia.\n");
    } else {
        while (finalPtr != NULL) {
            printf("%d\n", final -> num);
            final = final -> antPtr;
        }
    }

    return;
}
```

```c
void mostrarPreOrden(nodo *raiz) {

    if (raiz != NULL) {
        printf("%d - ", raiz -> num);
        mostrarPreOrden(raiz -> izqPtr);
        mostrarPreOrden(raiz -> derPtr);
    }
    return;
}


void mostrarEnOrden(nodo *raiz) {

    if (raiz != NULL) {
        mostrarEnOrden(raiz -> izqPtr);
        printf("%d - ", raiz -> num);
        mostrarEnOrden(raiz -> derPtr);
    }
    return;
}
void mostrarPostOrden(nodo *raiz) {

    if (raiz != NULL) {
        mostrarPostOrden(raiz -> izqPtr);
        mostrarPostOrden(raiz -> derPtr);
        printf("%d - ", raiz -> num);
    }
    return;
}
```

(5) Desacoplamiento.

```c
nodo desacoplarLista(nodo **lista) {
    nodo *nodoAuxiliar, desacoplado;

    desacoplado = **lista;
    nodoAuxiliar = *lista;

    *lista = (*lista)->sigPtr;
    free(nodoAuxiliar);

    return desacoplado;
}
```

```c
nodo desacoplarLista(nodo **lista, nodo **final) {
    nodo *nodoAuxiliar, desacoplado;

    desacoplado = **lista;
    nodoAuxiliar = *lista;

    *lista = (*lista)->sigPtr;

    if(*lista == NULL) {
        *final = NULL;
    } else {
        (*lista) -> antPtr = NULL;
    }
    free(nodoAuxiliar);

    return desacoplado;
}
```

(6) Eliminación (de un nodo).

```c
void eliminarNodo(nodo **cabeza, int dato) {
    nodo *nodoAuxiliar, *nodoAnterior, *nodoActual;

    if (*cabeza == NULL) {
        printf("La lista esta vacia.\n");
    } else {
        nodoAnterior = *cabeza;
        nodoActual = *cabeza;

        while (nodoActual != NULL && nodoActual -> num != dato) {
            nodoAnterior = nodoActual;
            nodoActual = nodoActual -> sigPtr;
        }

        if (nodoAnterior == nodoActual) {
            nodoAuxiliar = *cabeza;
            (*cabeza) = (*cabeza) -> sigPtr;
            free(nodoAuxiliar);
        } else if (nodoActual != NULL) {
            nodoAuxiliar = nodoActual;
            nodoAnterior -> sigPtr = nodoActual -> sigPtr;
            free(nodoAuxiliar);
        } else {
            printf("El nodo con el dato indicado no se encuentra en la lista.\n");
        }

    }
    return;
}
```

```c
void eliminarNodo(nodo **cabeza, nodo **final, int dato) {
    nodo *nodoAuxiliar, *nodoAnterior, *nodoActual;

    if (*cabeza == NULL) {
        printf("La lista esta vacia.\n");
    } else {
        nodoAnterior = *cabeza;
        nodoActual = *cabeza;

        while (nodoActual != NULL && nodoActual -> num != dato) {
            nodoAnterior = nodoActual;
            nodoActual = nodoActual -> sigPtr;
        }

        if (nodoAnterior == nodoActual) {
            nodoAuxiliar = *cabeza;
            *cabeza = (*cabeza) -> sigPtr;
            if (*cabeza == NULL) {
                *final == NULL;
            } else {
                (*cabeza) -> antPtr = NULL;
            }
            free(nodoAuxiliar);
        } else if (nodoActual != NULL) {
            nodoAuxiliar = nodoActual;
            nodoAnterior -> sigPtr = nodoActual -> sigPtr;
            if (nodoAnterior -> sigPtr == NULL) {
                *final = nodoAnterior;
            } else {
                (nodoAnterior -> sigPtr) -> antPtr = nodoAnterior;
            }
            free(nodoAuxiliar);
        } else {
            printf("El nodo con el dato indicado no se encuentra en la lista.\n");
        }

    }

    return;
}
```

(7) Eliminación (total).

```c
void eliminarLista(nodo **cabeza) {
    nodo *nodoAuxiliar;

    if (*cabeza == NULL) {
        printf("La lista esta vacia.\n");
    } else {
        while (*cabeza != NULL) {
            nodoAuxiliar = *cabeza;
            *cabeza = (*cabeza) -> sigPtr;
            free(nodoAuxiliar);
        }
    }
    return;
}
```

```c
void eliminarLista(nodo **cabeza, nodo **final) {
    nodo *nodoAuxiliar;

    if (*cabeza == NULL) {
        printf("La lista esta vacia.\n");
    } else {
        while (*cabeza != NULL) {
            nodoAuxiliar = *cabeza;
            *cabeza = (*cabeza) -> sigPtr;
            free(nodoAuxiliar);
        }
        *final = NULL;
    }

    return;
}
```

```c
void eliminarArbol(nodo **raiz) {
    nodo *nodoAuxiliar;

    if (*raiz != NULL) {
        eliminarArbol(&(raiz->izqPtr));
        eliminarArbol(&(raiz->derPtr));
        nodoAuxiliar = *raiz;
        *raiz = NULL;
        free(nodoAuxiliar);
    }
    return;
}
```