UNSA - CS - INTRODUCCIÓN A LA CIENCIA DE LA COMPUTACIÓN
Ernesto Cuadros
Kevin Salazar Torres | 20162013

LISTAS

```cpp
#include<iostream>

using namespace std;
using Key = int;
using Data = int;
using Size = int;

class Nodo
{
public:
    Key key;
    Data *data;
    Nodo *sgte;
    Nodo(Key key = 0, Data *data = NULL): key(key), data(data), sgte(NULL) {}
};

class Lista_E: public Nodo
{
public:
    Nodo *lista;
    Lista_E(): lista(NULL){}
    void print() const
    {
        if (lista == NULL)
            cout << "NULL" << '\n';
        else{
            cout << lista->key << " -> ";
            Nodo *current = lista->sgte;
            for(;current != NULL; current = current->sgte){
                cout << current->key << " -> ";
            }
            cout << "NULL" << '\n';
        }
    }
    Lista_E *insert(Key key, Data *data)
    {
        Nodo *nodo = new Nodo(key,data); //IMPORTANTE CREAR DINAMICAMENTE
        if (lista == NULL){
            lista = nodo;
            return this;
        }
        if (nodo->key <= lista->key){
            nodo->sgte = lista;
            lista = nodo;
            return this;
```

```
        }
        Nodo **ptr_current = &(lista->sgte);
        for(;*ptr_current != NULL && (*ptr_current)->key <= nodo->key;
ptr_current = &((*ptr_current)->sgte));
        nodo->sgte = *ptr_current;
        *ptr_current = nodo;
        return this;
    }
};

struct nodo *crear(int n)
{
    if (n>0){
        //cabeza de lista
        struct nodo *head = malloc(sizeof(struct nodo));
        head->numero = 7;
        head->siguiente = NULL;
        //lo demás
        if (n>1){
            struct nodo *cur = head; //tengo un current, ahora voy a agregar
            for (int i = 1; i != n; i++){
                struct nodo *nuevo = malloc(sizeof(struct nodo));
                nuevo->numero = i;
                nuevo->siguiente = NULL;
                cur->siguiente = nuevo;
                cur = nuevo;
            }
        }
        return head;
    }
    return NULL;
}

void imprimir(struct nodo *head)
{
    if (head == NULL)
        printf("NULL\n");
    else{
        //imprime cabeza
        printf("%d -> ",head->numero);
        //lo demás
        struct nodo *cur = head->siguiente; //tengo un current, ahora voy a
imprimir
        while (cur != NULL){
            printf("%d -> ",cur->numero);
            cur = cur->siguiente;
        }
        printf("NULL\n");
    }
}
void eliminar(struct nodo **dir_head, int posicion)
```

```
{
    if (*dir_head != NULL){
        //eliminar cabeza
        if (posicion == 0){
            struct nodo *temp = *dir_head;
            *dir_head = (*dir_head)->siguiente;
            free(temp);
        }
        // lo demás
        else{
            struct nodo *anterior = *dir_head;
            struct nodo *cur = (*dir_head)->siguiente;
            int i = 1;
            while ((cur != NULL) && (anterior == NULL || cur == NULL || i !=
posicion)){
                anterior = cur;
                cur = cur->siguiente;
                i++;
            }
            if (cur == NULL)
                return;
            else{
                anterior->siguiente = cur->siguiente;
                free(cur);
            }
        }
    }
}

void concatenar(struct nodo **dir_head1, struct nodo *head2)
{
    if (*dir_head1 == NULL){
        *dir_head1 = head2;
        return;
    }
    else{
        struct nodo *cur = *dir_head1;
        while (cur->siguiente != NULL)
            cur = cur->siguiente;
        cur->siguiente = head2;

    }
}

void agregar(struct nodo **dir_head, int numero, int posicion)
{
    struct nodo *nuevo = malloc(sizeof(struct nodo));
    nuevo->numero = numero;
    nuevo->siguiente = NULL;
    //agregar a cabeza
    if (posicion == 0){
```

```
        nuevo->siguiente = *dir_head;
        *dir_head = nuevo;
    }
    else{
        struct nodo *anterior = *dir_head;
        struct nodo *cur = (*dir_head)->siguiente;
        for(int i = 1; i != posicion; anterior = cur, cur = cur->siguiente, i++)
            ;
        anterior->siguiente = nuevo;
        nuevo->siguiente = cur;
    }

}



int main(void)
{

    Lista_E le;

    Data data;
    le.insert(0,&data)->insert(13,&data)->insert(8,&data)->insert(-7,&data)-
>insert(18,&data)->insert(8,&data);

    //cout<< (le.lista)->sgte->sgte->key;

    le.print();

      struct nodo *head = crear(5); // del 0 al 4
    imprimir(head);

    eliminar(&head,5); //el free, a un int lo vuelve cero y a un puntero lo deja
como esta (no lo pone a NULL)
    imprimir(head);

    struct nodo *head1 = NULL;
    concatenar(&head1, head);
    imprimir(head1);

    struct nodo *head3 = crear(3);
    concatenar(&head1, head3);
    imprimir(head1);

    agregar(&head, 9, 6);
    imprimir(head);

    return 0;
}
```

PILA, USANDO VECTOR, con ejemplo de grafo de listas enlazadas

```
#include <iostream>
#include <vector>
#include <map>
using Data = int;

using namespace std;
typedef unsigned int Size;

template <typename T>
class Stack;

template < typename T>
Stack <T> operator+( const Stack <T> &s1 , const Stack <T> &s2)//concatena pilas
{
    Stack <T> r = s1;
    for ( Size i = 0; i < s2.elementos.size(); ++i) {
        r.elementos.push_back(s2.elementos[i]);
    }
    return r ;
}

template <typename T>
class Stack
{
private:
    vector <pair<T,Data*>> elementos;  //el nodo es el key y un puntero al dato
public :
    bool empty () const
    {
        return elementos.empty();
    }
    void push( const T &e, const Data*d )
    {
        pair<T,Data*> par;
        par.first = e;
        par.second = d;
        elementos.push_back ( par );
    }
    pair<T,Data*> pop () {
        pair<T,Data*> par;
        par = elementos.back ();
        elementos.pop_back ();
        return par;
    }

    friend Stack <T> operator+( const Stack <T> &s1 , const Stack <T> &s2);
};

//////////////////grafo
class Graph
```

```
{
protected :
    map <int , vector <int > > mapa ;

public :
    Graph ( const vector <int > &origen , const vector <int > &destino )
    {
        for( Size i = 0; i < origen.size (); i++ ) {
            int o = origen[i];
            int d = destino[i];
            mapa[o].push_back(d);
        }
    }

    const vector <int > &adyacente( const int key ) const
    {
        map <int,vector <int > >:: const_iterator it = mapa.find(key);
        if(it !=  mapa.end()) {
            return it-> second ;
        }
        cerr << "No hay nodo";
    }

    int cant_nodos (const int key ) const {
        return adyacente(key).size();
    }
};
//////////////////////
int main()
{
    Stack<int> pila;
    Data *d;
    pila.push(0,d);
    pila.pop();
    if(pila.empty()) cout << "Bien\n";

    pila.push(1,d);
    pila.push(2,d);
    Stack<int> pila2;
    pila2.push(3,d);
    pila2.push(4,d);
    Stack<int> pila3;

    pila3 = pila + pila2;
    cout << pila3.pop().first;
    cout << pila3.pop().first;

    //para grafo
    vector<int> start;
    start.insert(start.begin(), 3,1);
    start.insert(++++++(start.begin()), 2,5);
```

```
    start.push_back(4);

    vector<int> endd;
    endd.push_back(2,d);
    endd.push_back(3,d);
    endd.push_back(4,d);
    endd.push_back(4,d);
    endd.push_back(2,d);
    endd.push_back(2,d);

    Graph g(start, endd);
    cout << g.cant_nodos(1 ) << '\n';
    vector<int> ad = g.adyacente(1);
    for(vector<int>::iterator it = ad.begin(); it != ad.end(); it++){
        cout << *it << ' ';
    }
    cout << '\n';

    return 0;
}
```

COLA, USANDO VECTOR

```
#include <iostream>
#include <vector>

using namespace std;
using Data = int;

template <typename T>
class Cola
{
private:
    vector <pair<T,Data*>> elementos;  //el nodo es el key y un puntero al dato
public :
    bool empty () const
    {
        return elementos.empty();
    }
    void push( const T &e, const Data*d )
    {
        pair<T,Data*> par;
        par.first = e;
        par.second = d;
        elementos.push_back ( par );
    }
    pair<T, Data*> pop () { //devuelve un par: llave, dato
        pair<T,Data*> par;
        par = elementos[0];
        elementos.erase(0);
        return par;
    }
```

```
};


int main()
{
    Cola<int> cola;
    Data* dato;
    pila.push(0,dato);
    pila.pop();

    return 0;
}
```

ARBOL BINARIO DE BÚSQUEDA

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100

struct tnode{
    char *word;              // apunta al contenido del arbol
    int count;
    struct tnode *left;
    struct tnode *right;
};

struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);
struct tnode *talloc(void);
struct tnode *balancear_arbol(struct tnode *);
struct tnode *arbol_a_lista(struct tnode*, struct tnode**);
int cantidad_nodos(struct tnode *);
struct tnode* lista_a_arbol(struct tnode**, int );
void pre_orden(struct tnode*);

int main(void){ //se leen palabras y se colocan en un arbol que se balancea
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
        root = addtree(root, word);
    treeprint(root);

    pre_orden(root);
    root = balancear_arbol(root);
    printf("\n");
    pre_orden(root);
```

```
    return 0;
}

struct tnode *balancear_arbol(struct tnode * root)
{
    struct tnode *cabeza = NULL; //cabeza de lista
    arbol_a_lista(root, &cabeza);
    int n = cantidad_nodos(cabeza);
    return lista_a_arbol(&cabeza,n);
}

struct tnode *arbol_a_lista(struct tnode* root, struct tnode** cabeza_dir)
{
    if (root == NULL)
        return NULL;
    arbol_a_lista(root->right, cabeza_dir);
    root->right = *cabeza_dir;
    if(*cabeza_dir != NULL)
        (*cabeza_dir)->left = root;
    *cabeza_dir = root;
    arbol_a_lista(root->left,cabeza_dir);
}

int cantidad_nodos(struct tnode * cabeza)
{
    int cant = 0;
    while (cabeza)
    {
        cabeza = cabeza->right;
        cant++;
    }
    return cant;
}

struct tnode* lista_a_arbol(struct tnode** cabeza_dir, int n)
{
    if (n<=0)
        return NULL;
    struct tnode* left = lista_a_arbol(cabeza_dir, n/2);
    struct tnode* root = *cabeza_dir;
    root->left = left;
    *cabeza_dir = (*cabeza_dir)-> right;
    root->right = lista_a_arbol(cabeza_dir, n-n/2-1);
    return root;
}

void pre_orden(struct tnode* root)
{
    if (root == NULL)
        return;
```

```
    printf("%s ", root->word);
    pre_orden(root->left);
    pre_orden(root->right);
}

struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL){               // nuevo contenido ha llegado
        p = talloc();            // hacer nuevo nodo
        p->word = strdup2(w);
        p->count = 1;
        p->left = p->right = NULL;
    }
    else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;               // contenido repetido
    else if (cond < 0)
        p->left = addtree(p->left, w);
    else
        p->right = addtree(p->right, w);
    return p;
}
//inorder
void treeprint(struct tnode *p){
    if (p != NULL){
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}

#include <stdlib.h>

struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}

char *strdup2(char *s)    // duplicar s
{
    char *p;

    p = (char *) malloc(strlen(s)+1);   // +1 para '\0'
    if (p != NULL)
        strcpy(p, s);
    return p;
}

int getword(char *word, int lim)
{
```

```
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)){
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())){
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}

#define BUFFSIZE 100
 // se maneja adecuadamente el buffer
char buf[BUFFSIZE];
int bufp = 0;

int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)  // tomado el libro de C, el ungetch
{
    if (bufp >= BUFFSIZE)
        printf("ungetch: too many characters\n");
    else buf[bufp++] = c;
}
```