



Lección 9:

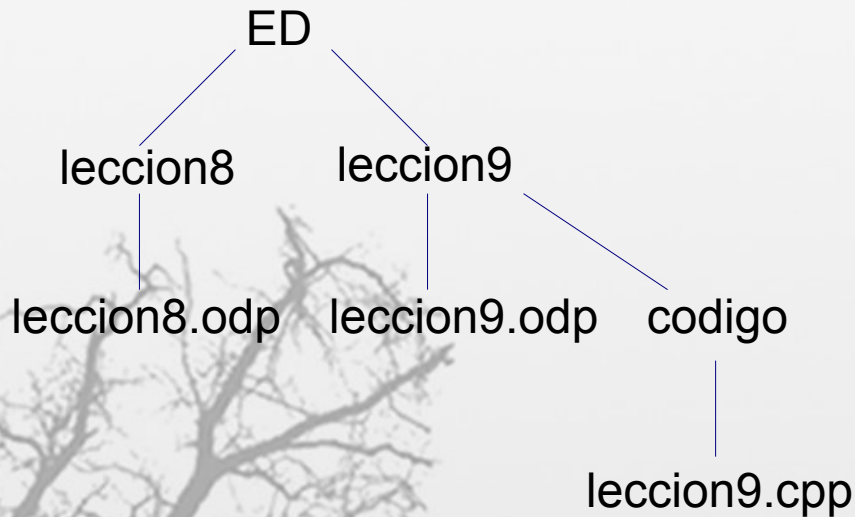
Pilas, colas y colas con prioridad

- Motivación
- Pilas: impl. mediante vectores y listas enlazadas
- Colas: impl. mediante vectores y listas
- Colas con prioridad: impl. mediante listas ordenadas, vectores de listas y listas de listas
- Pilas, colas y colas con prioridad en STL
- Eliminación de recursividad

Motivación



Problema: para una aplicación que estamos implementando necesitamos una función que obtenga la lista archivos y directorios a partir de un directorio de partida



ED
ED/leccion8
ED/leccion8/leccion8.odp
ED/leccion9
ED/leccion9/leccion9.odp
ED/leccion9/codigo
ED/leccion9/codigo/leccion9.cpp

Motivación



- El siguiente código C++ resuelve el problema utilizando **recursividad**

```
#include <dirent.h>
#include <vector>
using namespace std;


void recorrerDirRec(
    const string &dir,
    vector<string> &listaEntradas)
{
    DIR *pdir;
    struct dirent *pent;

    pdir=opendir(dir.c_str());
    if (pdir){
        string entrada;
```

```
        while ((pent=readdir(pdir))){
            entrada = pent->d_name;

            if (entrada!="." && entrada!="..") {
                listaEntradas.push_back(
                    dir + "/" + entrada);

                if (pent->d_type == DT_DIR) {
                    recorrerDirRec(
                        string(listaEntradas.back()),
                        listaEntradas);
                }
            }
        }
        closedir(pdir);
    }
}
```

 **Llamada recursiva**

Motivación (cont.)



- La recursividad es una técnica muy poderosa pero tiene serios inconvenientes:
 - Genera una gran cantidad de llamadas a funciones (ineficiente)
 - Puede consumir mucha memoria y desbordar la pila de la aplicación

En el ejemplo anterior hay un inconveniente adicional. Las llamadas recursivas van abriendo nuevos directorios sin cerrar los anteriores, pero el número de directorios que se pueden mantenerse abiertos simultáneamente con la función *opendir()* es limitado, por lo que puede desbordarse fácilmente

Motivación (cont.)



- En muchos casos una función recursiva puede convertirse en no recursiva con ayuda de una pila para guardar resultados intermedios
- Esta implementación es mucho más estable, rápida y consume menos memoria

Veremos al final de la lección como transformar esta función en no recursiva



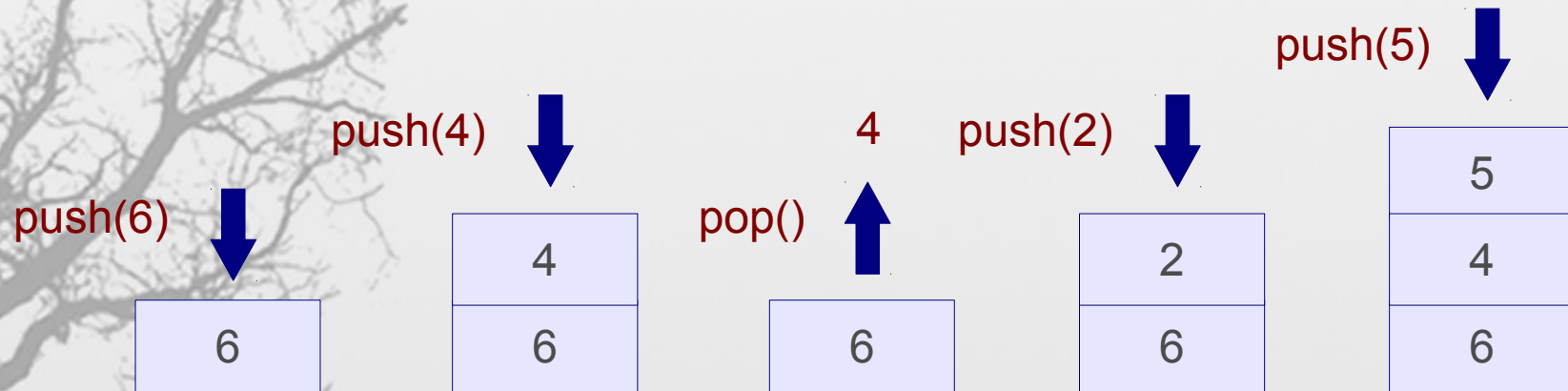
Pilas, colas y colas con prioridad

- Las **pilas** y **colas** son las primeras estructuras de datos que se inventaron y son la más sencillas
- Omnipresentes en los sistemas informáticos, tanto hardware como software
- Las colas con prioridad son estructuras de datos más complejas
- En general no tienen una implementación propia y se montan sobre otras estructuras de datos como vectores o listas enlazadas

Pilas



- Una pila es una estructura de datos con tres operaciones básicas: *push()*, *pop()* y *top()*
 - *push()* introduce un elemento en la pila
 - *pop()* extrae el siguiente elemento
 - *top()* devuelve el siguiente elemento (sin sacarlo)
- El orden de salida es inverso al de entrada, por eso se llaman también LIFOs (last in/first out)



Implementación mediante vectores



- Implementar una pila mediante un vector estático (automático o en memoria dinámica) es trivial
- Basta con añadir y quitar datos por el final

```
template<class T>
class Pila {
    T *datos;
    int numDatos;

public:
    Pila(int max) {
        datos = new T[max];
        numDatos = 0;
    }

    void push(T &dato) {
        datos[numDatos++] = dato;
    }
}
```

```
T pop() {
    return datos[--numDatos];
}

T top() {
    return datos[numDatos - 1];
}

bool vacia() {
    return (numDatos != 0);
}
```


Implementación mediante vectores dinámicos



- Más flexible al no tener que especificar el tamaño máximo desde el principio

```
#include <vector>
using namespace std;

template<class T>
class PilaDin {
    vector<T> datos

public:
    Pila(int max): datos() {}

    void push(T &dato) {
        datos.push_back(dato)
    }
}
```

```
T pop() {
    T dato = datos.back();
    datos.pop_back();
    return dato;
}

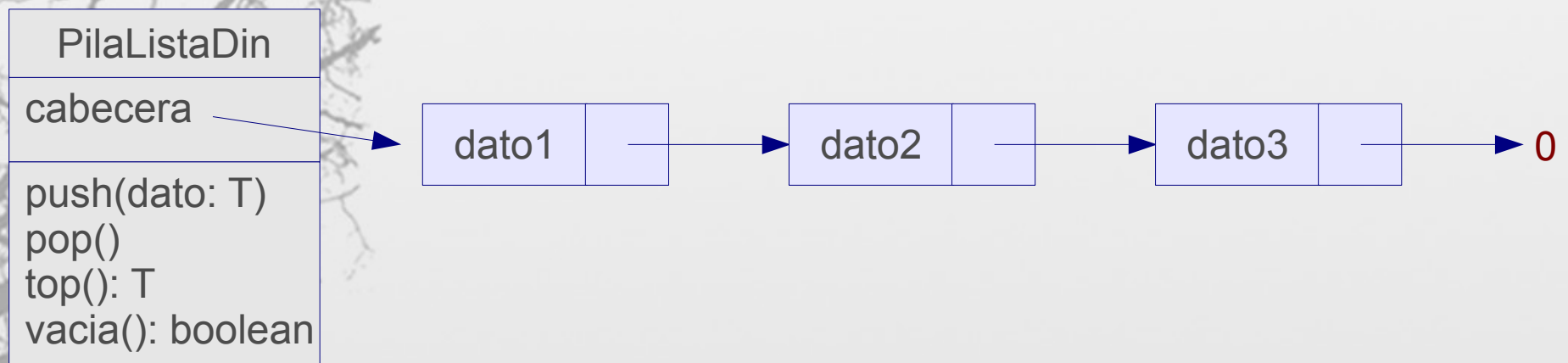
T top() {
    return datos.back();
}

bool vacia() {
    return datos.empty();
}
}
```

Implementación mediante listas enlazadas



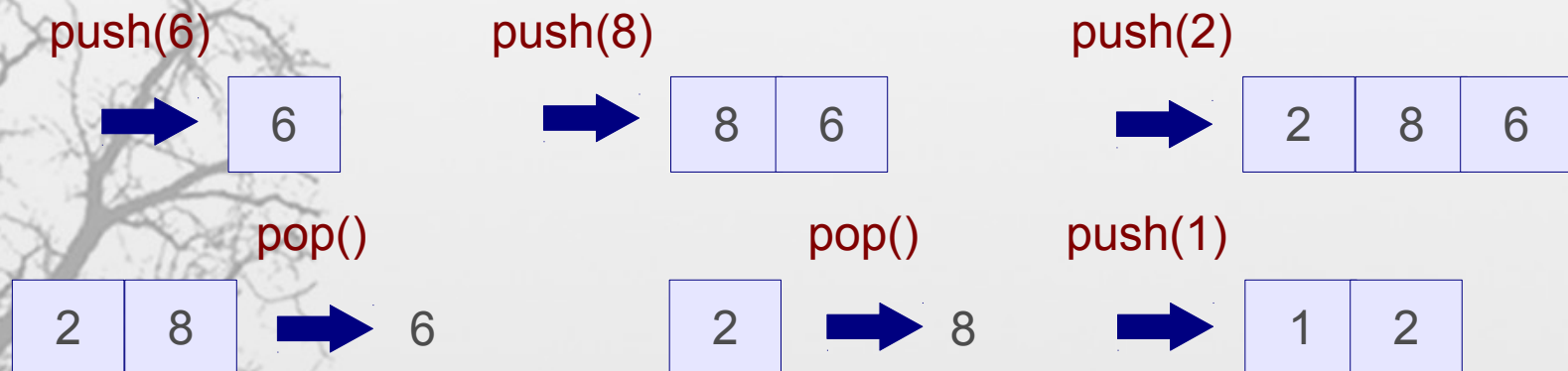
- También puede implementarse como una lista simplemente enlazada, metiendo y sacando por el principio
- Puede usarse una lista enlazada existente o implementarse directamente (sólo es necesario el puntero al principio)



Colas



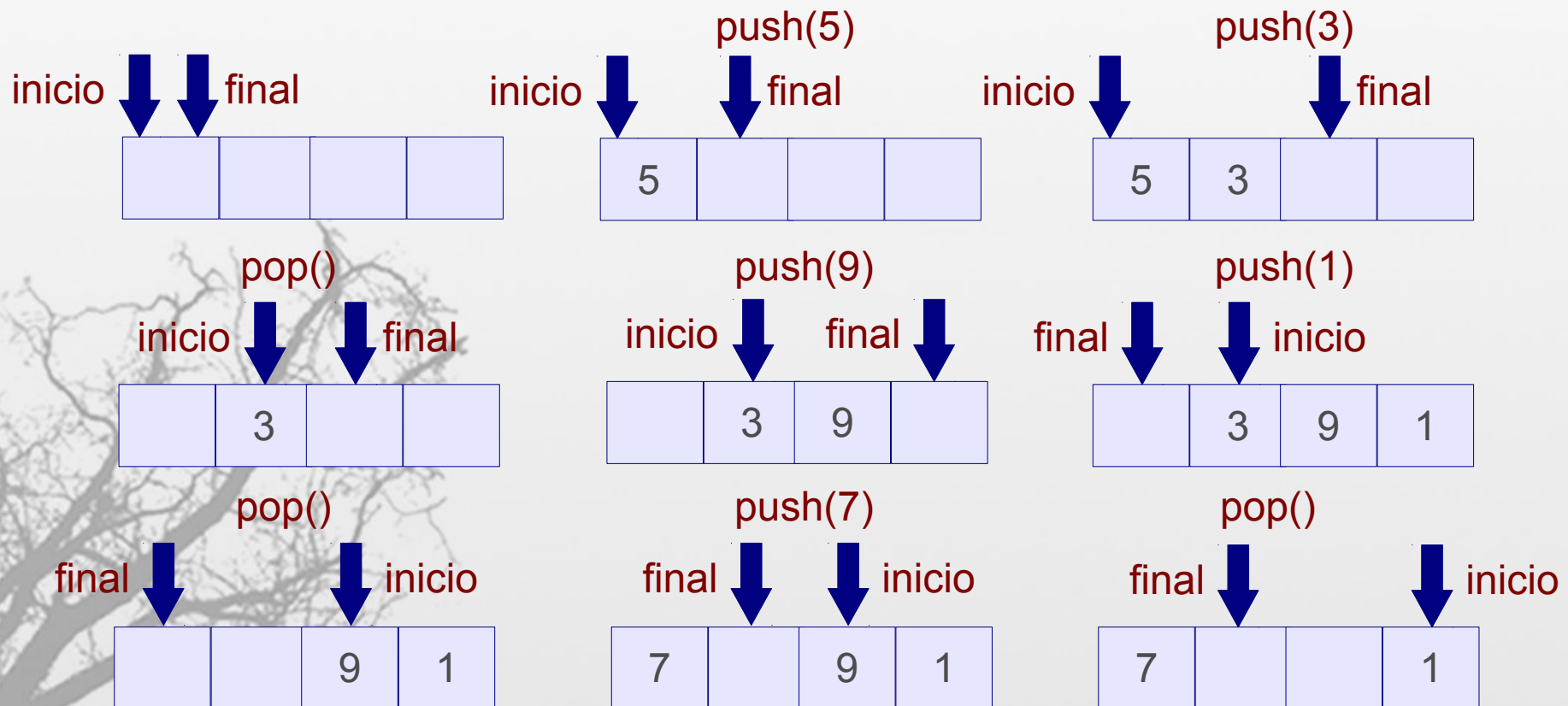
- Similar a una pila, pero el orden de entrada de los elementos es igual al de salida, por eso se también se llaman FIFOs (first in/first out)
- Se utilizan principalmente como lista de espera para atender peticiones o para el acceso a un recurso compartido



Implementación mediante vectores



- Se implementa con dos índices (inicio y final) que avanzan siguiendo un esquema circular
- Los push se hacen al final y los pop al principio



Implementación mediante vectores



- La implementación es muy sencilla:

```
template<class T>
class Cola {
    T *datos;
    int inicio, final;
    int tamMax;

public:
    Cola(int aTamMax) {
        datos = new T[aTamMax];
        tamMax = aTamMax;
        inicio = final = 0;
    }

    void push(T &dato) {
        datos[final] = dato;
        final = (final + 1) % tamMax;
    }
}
```

```
T pop() {
    T dato = datos[inicio];
    inicio = (inicio + 1) % tamMax;
}

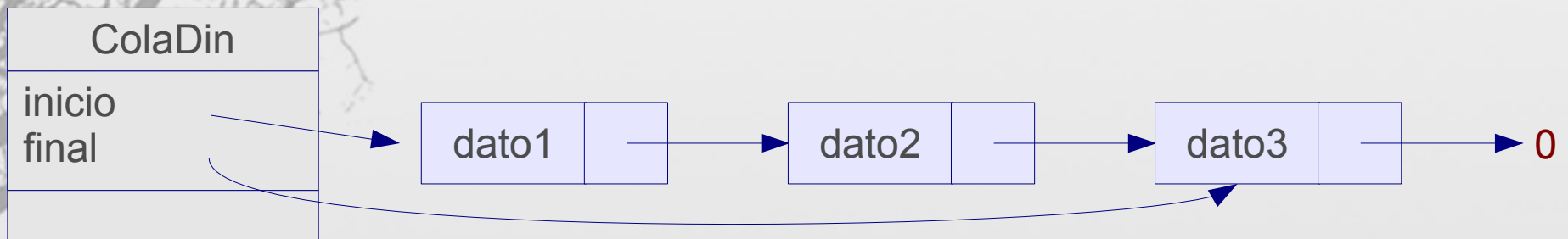
T top() {
    return datos[inicio];
}

bool vacia() {
    return (inicio == final);
}
```

Colas dinámicas



- Si no conocemos el tamaño máximo a priori, el esquema circular anterior no sirve
- No puede usarse un vector dinámico porque la inserción/borrado por el principio tiene un coste alto
- Puede usarse un deque, aunque la implementación más habitual es usando una lista simplemente enlazada (realizada ex profeso o no)
- Los push se realizan insertando al final y los pop borrando al inicio



Colas con prioridad



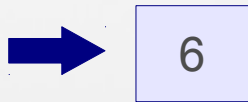
- Una cola donde cada dato tiene una prioridad
- El siguiente dato en salir es el de prioridad más alta
- Al igual prioridad, por orden de llegada
- Sirve para organizar el acceso a un recurso escaso por parte de un conjunto de solicitantes de distinta importancia

Una cola de impresión en una organización corporativa puede funcionar como una cola con prioridad. Se van aceptando trabajos de impresión de todos los empleados pero siempre atendiendo en primer lugar a los de mayor prioridad

Ejemplo de funcionamiento

- Si al hacer push insertamos el elemento ordenando por prioridad, el pop siempre devuelve el elemento correcto

push(6)



push(8)



push(2)



pop()



push(7)



pop()



Lista enlazada ordenada



- Implementación trivial: una lista simplemente enlazada ordenada por prioridad
- Pop → borrado por el inicio: $O(1)$
- Push → inserción ordenada: $O(n)$

```
#include <list>
using namespace std;

template<class T>
class ColaPri {
    list<T> datos;

public:
    ColaPri(): datos() {}

    T top() {
        return datos.front();
    }
}
```

```
T pop() {
    T dato = datos.front();
    datos.pop_front();
    return dato;
}

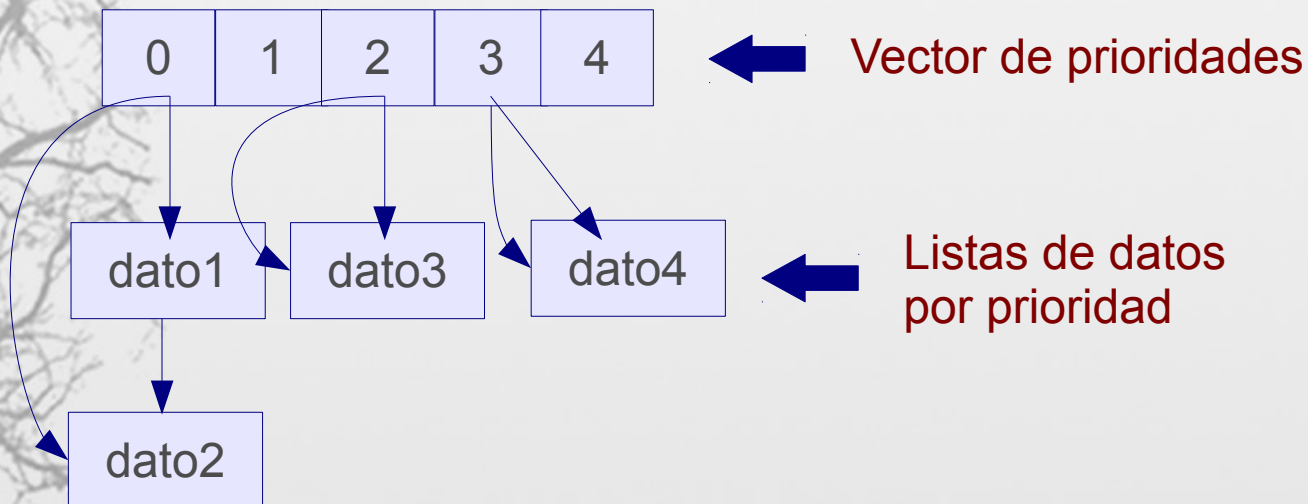
void push(T &dato) {
    list<T>::iterator i = datos.begin()
    while (i != datos.end() &&
           *i <= dato)
        ++i;
    datos.insert(i, dato);
}
...
```

← Comparación
por prioridad

Vectores de listas



- La implementación anterior es sólo para un número moderado de datos. En caso contrario es necesario recurrir a un **heap** (lección 12)
- Si existe la posibilidad de codificar las prioridades (p. e. con un número) y su rango es pequeño existe una implementación más eficiente:



Vectores de listas (cont.)

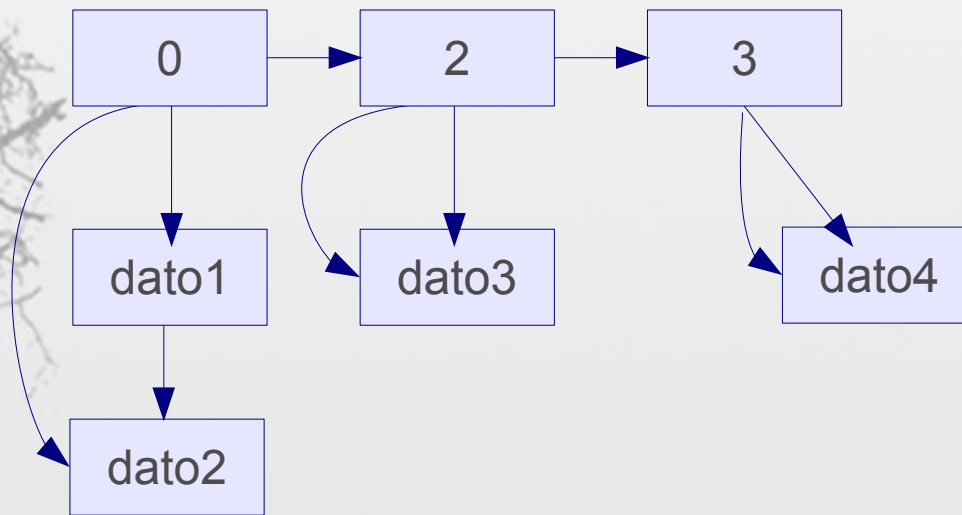


- Los push requieren tiempo $O(1)$:
 - Localizar la prioridad del dato en el vector \rightarrow tiempo $O(1)$ (no depende del número de datos almacenados)
 - La inserción al final de la lista $\rightarrow O(1)$
- Los pop requieren también tiempo $O(1)$:
 - Acceder a la primera posición del vector \rightarrow tiempo $O(1)$
 - Sacar de la lista $\rightarrow O(1)$

Listas de listas



- Si no se pueden mantener todas las prioridades posibles en el vector, puede usarse un vector dinámico y mantener sólo las que se están usando
- En este caso es preferible usar una lista para las prioridades, puesto que las inserciones y borrados son más eficientes



Pilas, colas y colas con prioridad en STL



- No son considerados contenedores sino **adaptadores** de contenedores, puesto que se implementan sobre un contenedor base
- El contenedor base se indica como parámetro, aunque ya tienen uno asignado por defecto (*deque* para pilas y colas y *vector* para colas con prioridad)
- No soportan iteración
- Pilas → clase *stack*
- Colas → clase *queue*

Ejemplo



```
#include <stack>
using namespace std;

int main() {
    stack<char> pila;
    char c;

    for (c = 'A'; c <= 'Z'; c++)
        pila.push(c);

    while (!pila.empty()) {
        c = pila.top();
        pila.pop();

        cout << c << " ";
    }

    return 0;
}
```

```
#include <queue>
#include <list>
using namespace std;

int main() {
    // Cola construida sobre lista
    queue<char, list<char> > cola;
    char c;

    for (c = 'A'; c <= 'Z'; c++)
        cola.push(c);

    while (!cola.empty()) {
        c = cola.top();
        cola.pop();

        cout << c << " ";
    }

    return 0;
}
```

Colas con prioridad



- La clase ***priority_queue*** está construida sobre un *heap*, que a su vez se monta por defecto sobre un vector
- Requieren una **clase de comparación** para ordenar los datos
- Las clases de comparación predefinidas de STL son suficientes para la mayoría de los casos:
 - *less<T>* → compara los datos usando el operador <
 - *greater<T>* → compara los datos usando el operador >
- Por defecto *priority_queue* usa *less<T>* como función de comparación

Clases de comparación predefinidas



- Para clases complejas podemos definir nuestra propia clase de comparación

Vamos a construir una clase de comparación para ordenar los trabajos de una cola de impresión, dando prioridad a trabajos pequeños sobre grandes

```
class TrabajoImpresion {
    string usuario; // Usuario que envia el trabajo
    long tam; // Tamaño del trabajo
};

// Dar prioridad a trabajos pequeños sobre grandes
class TrabajoMayorPrioridad {

    // Hay que devolver true si a tiene más prioridad que b
    bool operator()(TrabajoImpresion &a, TrabajoImpresion &b) {
        return a.tam < b.tam;
    }
};

// Instanciación
priority_queue<TrabajoImpresion, TrabajoMayorPrioridad> trabajos;
```


Clases de comparación predefinidas



- Para clases complejas podemos definir nuestra propia clase de comparación

Vamos a construir una clase de comparación para ordenar los trabajos de una cola de impresión, dando prioridad a trabajos pequeños sobre grandes

```
class TrabajoImpresion {
    string usuario; // Usuario que envia el trabajo
    long tam; // Tamaño del trabajo
};

// Dar prioridad a trabajos pequeños sobre grandes
class TrabajoMayorPrioridad {
    // Hay que devolver true si a tiene más prioridad que b
    bool operator()(TrabajoImpresion &a, TrabajoImpresion &b) {
        return a.tam < b.tam;
    }
};

// Instanciación
priority_queue<TrabajoImpresion, list<TrabajoImpresion>,
TrabajoMayorPrioridad> trabajos;
```

Ejemplo:



```
#include <queue>

int main() {
    // Por defecto usa un vector como contenedor
    // y ordena de menor a mayor con less<T>
    priority_queue<int> colaPri;

    colaPri.push(7);
    colaPri.push(2);
    colaPri.push(4);
    colaPri.push(8);

    // Imprime los datos por este orden: 2, 4, 7, 8
    while (!colaPri.empty()) {
        cout << colaPri.top() << endl;
        colaPri.pop();
    }

    return 0;
}
```

Solución al problema



```
void recorrerDir(
    const string &dir,
    vector<string> &listaEntradas)
{
    DIR *pdir;
    struct dirent *pent;

    // Pila de directorios
    stack<string> listaDir;
    string siguienteDir;

    listaDir.push(dir);

    while (!listaDir.empty()) {
        siguienteDir = listaDir.top();
        listaDir.pop();

        pdir = opendir (
            siguienteDir.c_str()
        );
        if (pdir){
            string entrada;
```

```
            while ((pent=readdir(pdir))){
                entrada = pent->d_name;
                if (entrada != "." &&
                    entrada != "..") {

                    listaEntradas.push_back(
                        siguienteDir +
                        "/" + entrada
                    );

                    if (pent->d_type == DT_DIR) {
                        listaDir.push(
                            string(listaEntradas.back())
                        );
                    }
                }
            }
            closedir(pdir);
        }
    }
}
```

Eliminación de recursividad

- No en todos los casos puede eliminarse la recursividad
- Patrón para eliminación de recursividad:
 - Meter los parámetros iniciales en la pila
 - Crear una bucle mientras haya datos en la pila
 - Dentro del bucle cada llamada a la función se sustituye por meter los parámetros de la llamada en la pila
 - Dentro del bucle cada ejecución de la función se sustituye por sacar datos de la pila y realizar los cálculos

Conclusiones



- Las pilas y colas son estructuras de datos muy sencillas que pueden ser implementadas las EEDD ya estudiadas
- Las colas con prioridad son algo más complejas. Pueden ser implementadas con limitaciones mediante vectores de listas y listas de listas
- En STL estas tres estructuras de datos se denominan adaptadores, ya que se construyen sobre otras EEDD, y se denominan stack, queue y priority_queue
- Una de las aplicaciones más interesantes de las pilas es la eliminación de la recursividad