Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 10
Estuturas de Dados: Listas, Pilhas e Filas

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em Cortez 2011 , Reis, Rocha e Faria, 2007)

Escola de Engenharia
Universidade do Minho

Tipos de Dados Abstractos

- **Estruturas Lineares:**
  - definição
  - pilhas (*stacks)*
    - vector, list
    - Aplicações
    - implementação na STL (*Standard Template Library)*
  - filas (*queues)*
    - vector, list
    - Aplicações
    - implementação na STL (*Standard Template Library)*
  - listas (*lists)*
    - iteradores
    - vector, list
    - Aplicações
    - implementação na STL (*Standard Template Library)*

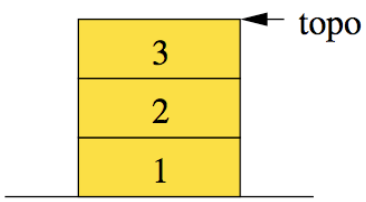
Tipos de Dados Abstractos

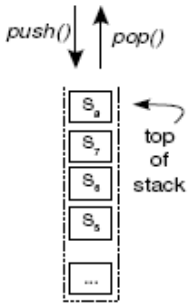
Definição de Tipo de Dados Abstractos (*sigla inglesa: ADT)*

- **conjunto de objectos + conjuntos de operações**
- **abstracção matemática (dados são genéricos e não específicos)**
- **operações especificadas são implementadas de forma genérica**
- **implementação em C++:**
  - classes genéricas
  - operações: membros-função públicos
- **Exos. de ADT: listas de objectos, filas, dicionários...**
- **Exos. de operações: comparar objectos, encontrar um objecto...**

Pilhas (Stacks)

- **Estruturas eficientes para armazenar e retirar dados na ordem LIFO - LAST IN FIRST OUT (último a entrar, primeiro a sair);**
- **Aplicações chamada de funções, recursividade, calculadoras, …**
- **estrutura de dados linear em que inserção e a remoção de elementos se faz pela mesma extremidade, designada por topo da pilha**
- **uma pilha pode ser considerada como uma *restrição de lista***
- **porque é uma estrutura de dados mais simples que a lista, é possível obter implementações mais eficazes**





Escola de Engenharia
Universidade do Minho

Pilhas (Stacks)

Manipulação (operações/métodos) sobre pilhas:

- **peek - inspeciona qual o elemento no topo, não retira**
- **pop - remove elemento do topo**
- **push - inclui elemento no topo**
- **empty - determina se a pilha está vazia.**
- **size - devolve o número de elementos**
- **init (construtor) - cria a pilha**
- **destroy (destrutor) - elimina a pilha**

Implementação:

- **Uma pilha pode ser implementada de diversas formas: vector, vector dinâmico, listas ligadas, etc…**
- **No exemplo utiliza-se pilha de inteiros (int) implementada utilizando um vector dinâmico**

Escola de Engenharia
Universidade do Minho

Pilhas (Stacks)

```
// exemplo completo, num único ficheiro stack.h
#include <iostream>
using namespace std;

class Stack { // uma stack de inteiros
    int top, d_size, *items; // topo, comprimento do vector dinamico
public:
    Stack(int size) { // construtor
        top = -1; d_size = size;
        items = new int[size];
    }
    ~Stack() { delete []items;}  // destrutor

    bool empty() const { return top==-1; } // O(1), true se stack vazia

    int size() const { return top+1; } // O(1), devolve o tamanho

    void push(int elem) { // O(1), inclui elem no topo
        if(size()==d_size) cerr << "Stack Full!\n";
        else { top++; items[top] = elem; }
    }
```

Pilhas (Stacks)

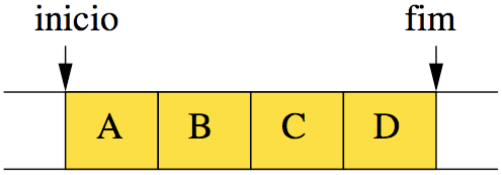
```
// continuação do stack.h
    int pop() { // O(1), remove e retorna o elem do topo
        if(this->empty()) cerr << "Empty stack\n";
        else return items[top--];
        return 0;
    }

    int peek() const { // O(1), inspeciona o topo
        if(this->empty()) cerr << "Empty stack\n";
        else return items[top];
        return 0;
    }

    void print() const { // O(n), mostra a pilha no ecra
        cout << "stack with " << size() << " elements\n";
        for(int i = size()-1; i>=0; i--)
            cout << items[i] << "\n";
        cout << "---“ << endl;
    }
}; // final da classe Stack
```

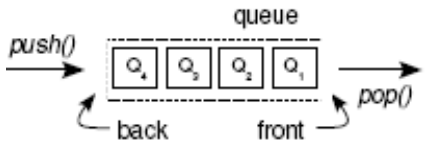

Classe stack (STL)

- **Alguns métodos da classe stack (STL):**
  - stack()
  - stack(const stack &)
  - bool empty() const
  - size_type size() const
  - T & top()
  - void push(const T &)
  - void pop()
  - stack & operator =(const stack &)
- **Funções globais (não membros da classe stack):**
  - bool operator ==(const stack &, const stack &)
  - bool operator <(const stack &, const stack &)
- **Ver:**
  - http://www.sgi.com/tech/stl/stack.html
  - http://www.cppreference.com/cppstack

Filas (Queues)

- **Estruturas eficientes para armazenar e retirar dados na ordem FIFO - FIRST IN FIRST OUT (primeiro a entrar, primeiro a sair)**
- **Aplicações: buffers, simulação de filas de espera em supermercados, …**
- **Estrutura de dados linear em que:**
  - inserção de elementos se faz por uma extremidade, designada por cauda
  - remoção de elementos se faz por extremidade oposta à cauda, designada por cabeça
- **uma fila pode ser considerada como uma *restrição de lista***





Filas (Queues)

Manipulação (operações/métodos) sobre filas:

- **peek - inspeciona qual o elemento no início, não retira**
- **dequeue - remove elemento do início**
- **enqueue - inclui elemento no fim**
- **empty - determina se a fila está vazia.**
- **size - devolve o número de elementos**
- **init (construtor) - cria a fila**
- **destroy (destrutor) - elimina a fila**

Implementação:

- **Uma fila pode ser implementada de diversas formas: vector, vetor dinâmico, listas ligadas, etc…**
- **Nesta aula, iremos utilizar uma fila de inteiros (int) via um vetor dinâmico…**
- **O vector é utilizado de modo circular, existindo dois registos especiais: início e fim…**

Filas (Queues)

```
// queue.h
#include <iostream>
using namespace std;

class Queue { // Fila (queue) de int
    int *items; // vector dinamico
    int inicio, fim, d_size;
public:
    Queue(int size) { //construtor
        d_size = size;
        items = new int[size];
        inicio = d_size-1;
        fim = d_size-1;
    }
    ~Queue() { // destrutor
        delete [] items;
    }
    bool empty() const { // O(1)
        return (inicio==fim);
    }
```


Filas (Queues)

```
// continuacao do queue.h
  int size() const { // O(1), comprimento da queue
        if (inicio<=fim) return fim-inicio;
        else return (d_size-inicio)+fim;
  }

  void enqueue(int elem) { // O(1), insere no fim
      int aux = fim;
      if(aux==d_size-1) aux=0; else aux++;
      if(inicio==aux) cerr << "Queue is full\n";
      else { fim=aux; items[aux]=elem;}
  }

  int dequeue() { // O(1), remove e devolve o elemento do inicio
      if (empty()){ cerr << "Queue is empty\n"; return 0;}
      else{
          if(inicio==d_size-1) inicio=0;
          else inicio++;
          return items[inicio];
      }
  }
```

Filas (Queues)

```
  int peek() const { // O(1), devolve o elemento no inicio
      if (empty()) { cerr << "Queue is empty\n"; return 0;}
      else {
           int aux;
           if(inicio==d_size-1) aux=0;
           else aux = inicio+1;
           return items[aux];
      }
  }

  void print() const { // O(n), imprime uma queue
      int i,j; int s=size();
      cout << "head: " << inicio << " end:" << fim
           << " size: " << s << " [";
      for(i=inicio,j=0; j<s; i++,j++) {
          if (i==d_size-1) i=-1;
          cout << items[i+1] << " ";
      }
      cout << "]\n";
  }
}; // fim de queue.h
```


Filas (Queues)

```
int main()
{
    Queue q(3);
    q.enqueue(3);
    q.enqueue(2);
    q.enqueue(1);
    q.print();
    cout << q.dequeue() << "\n";
}
```

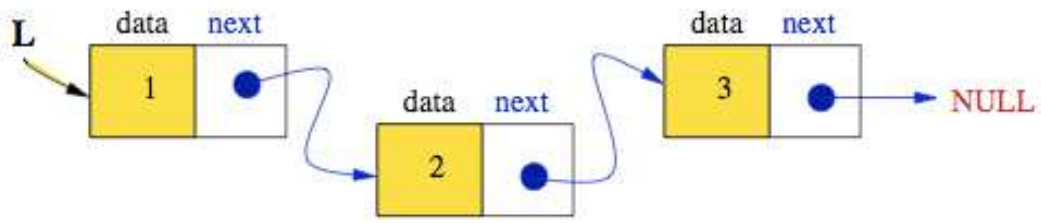
O Que aparece no écran?

Classe queue (STL)

- **Alguns métodos da classe queue (STL):**
  - queue()
  - queue(const queue &)
  - bool empty() const
  - size_type size() const
  - T & front()
  - T & back()
  - void push(const T &)
  - void pop()
  - queue & operator =(const queue &)
- **Funções globais (não membros da classe queue):**
  - bool operator ==(const queue &, const queue &)
  - bool operator <(const queue &, const queue &)
- **Ver:**
  - http://www.sgi.com/tech/stl/queue.html
  - http://www.cppreference.com/cppqueue

Listas Ligadas

- **Estrutura de memória dinâmica para dados sequenciais, organizados por nodos**
- **Mais eficiente para um uso intensivo de inserções e remoções**
- **Menos eficiente para aceder a um dado elemento: O (n)**
- **Lista vazia, não tem nodos (0)**



Listas Ligadas

Manipulação (operações/métodos) sobre listas:

- **remove - remove elemento da lista (se ele existir) - O(n)**
- **find – procura o nodo do elemento na lista – O(n)**
- **insert_head - inclui elemento no início da lista – O(1)**
- **insert_end – inclui elemento no final da lista – O(n)**
- **insert_sort – insere elemento de modo ordenado – O(n)**
- **empty - determina se a lista está vazia – O(1)**
- **size - devolve o número de elementos – O(n)**
- **init (construtor) – cria uma lista vazia – O(1)**
- **destroy (destrutor) – elimina toda a lista ligada – O(n)**

Implementação:

- **Existem 2 classes: Node e List**
- **Node: contém o atributo d_data e d_next, d_data será do tipo int embora também possa ser string ou uma classe qualquer…**
- **List: contém o atributo first, do tipo Node* e que aponta para o primeiro nodo da lista…**

Listas Ligadas

```
// exemplo completo, num único ficheiro list.h
#include <iostream>
using namespace std;
class Node // nodo de uma lista ligada de int
{ private:
    int d_data;   // pode ser string ou até um outro objecto
    Node *d_next; // aponta para o próximo nodo
public:
    Node(int data, Node *next){ d_data=data; d_next=next; }
    void setData(int data) { d_data=data; }
    void setNext(Node *next) { d_next=next; }
// acessores
    int data() const { return d_data;}
    Node *next() const { return d_next;}
    void print() const { cout << d_data << " ";}
};
```

Listas Ligadas: Implementação

- **Travesia – O(n):**
  - percorrer todos elementos da lista
- **Algoritmo típico:**

```
Node *aux=first;
while(aux!=0) {
    // processamento
    aux = aux->next();
}
```

- **Remover:**
  - procurar elemento, refazer ligações e apagar elemento

Listas Ligadas: Implementação

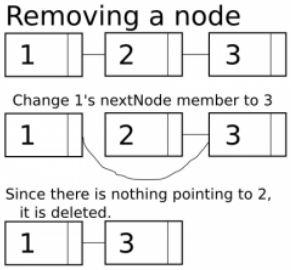

```
class List { // lista ligada de int
private:
    Node *first; // aponta para o primeiro nodo
public:
    List(){ // construtor de lista ligada vazia
        first=0;
    }
    ~List() { // destrutor
        Node *aux=first; Node *cur;
        while(aux!=0) {
            cur = aux; aux = aux->next(); delete cur;
        }
    }
    bool empty() const { return (first==0);}   // O(1)

    void print() const {   // O(n)
        Node *aux=first;
        while(aux!=0) { aux->print(); aux=aux->next();}
        cout << "\n";
    }
```

Listas Ligadas: Implementação

```
Node *find(int elem) { // O(n)
    Node *aux = first;
    while(aux!=0 && aux->data()!=elem) aux=aux->next();
    if(aux!=0) return aux;
    else { cerr << "Elem is not in list\n"; return 0;}
}

void insert_head(int elem){ // O(1), insere no inicio da lista
    Node *res = new Node(elem, first);
    first = res;
}

void insert_end(int elem){ // O(n), insere no fim
    Node *node, *aux;
    node = new Node(elem,0);
    aux = first;
    if(aux==0) first=node;
    else {
        while(aux->next()!=0) aux = aux->next();
        aux->setNext(node);
    }
}
```


Listas Ligadas: Implementação

```
void insert_sort(int elem) { // O(n), insere de modo ordenado
  Node *prev, *node, *aux;
  node = new Node(elem,0); prev = 0; aux = first;
  while(aux!=0 && aux->data()<elem) { prev = aux; aux = aux->next();}
  node->setNext(aux);
  if(prev==NULL) first=node; else prev->setNext(node);
}

Node *remove(int elem) { // remove elem da lista
  Node *prev, *aux;
  prev= 0; aux = first;
  while(aux!=0 && aux->data()!=elem){ prev = aux; aux = aux->next();}
  if(aux==0) cerr << "Item not found\n";
  else {
      if(prev!=0) prev->setNext(aux->next());
      else first=aux->next();
      delete aux;
  }
}
}; // fim de list.h
```



Listas Ligadas: Implementação

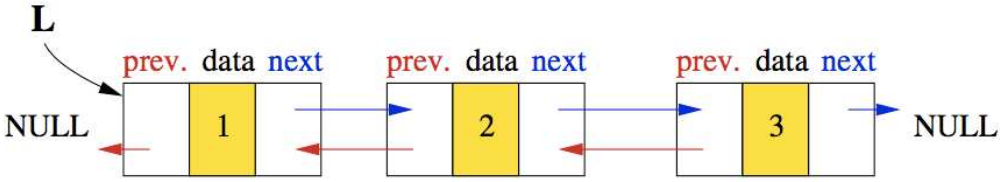
```
#include "list.h"
int main()
{
    List L;
    L.insert_sort(9);
    L.insert_sort(7);
    L.insert_sort(8);
    L.insert_head(1);
    L.insert_head(2);
    L.insert_head(3);
    L.insert_end(4);
    L.insert_end(5);
    L.insert_end(6);
    L.print();
    return 0;
}
```

O Que aparece no écran?

Listas Duplamente Ligadas

Lista Duplamente Ligada (*Double Linked List):*

- **Permite uma navegação em frente e reversa (para trás)**
- **Por exemplo, a remoção torna-se mais intuitiva**
- **Cada nodo armazena também o nodo anterior (previous);**



Listas Duplamente Ligadas

Manipulação (operações/métodos) sobre listas:

- remove - remove elemento da lista (se ele existir) - O(n)
- find – procura o nodo do elemento na lista – O(n)
- insert_head - inclui elemento no início da lista – O(1)
- insert_end – inclui elemento no final da lista – O(n)
- insert_sort – insere elemento de modo ordenado – O(n)
- empty - determina se a lista está vazia – O(1)
- size - devolve o número de elementos – O(n)
- init (construtor) – cria uma lista vazia – O(1)
- destroy (destrutor) – elimina toda a lista ligada – O(n)

Implementação (no exemplo):

- Existem 2 classes: DNode e DList;
- Node: contém o atributo d_data, d_next, d_previous e d_data (será do tipo string, mas pode ser uma qualquer classe…
- DList: contém os atributos first e last, do tipo DNode* e que apontam para o primeiro e último nodo da lista…

Listas Duplamente Ligadas

```
#include <iostream>
#include <string>
using namespace std;

class DNode // nodo de uma lista duplamente ligada
{
  private:
    string d_data; // pode ser uma outra classe
    DNode *d_next; DNode *d_previous;
  public:
    DNode(string data, DNode *previous, DNode *next) {
        d_data = data; d_previous = previous; d_next = next;
    }
    void setData(string data) { d_data = data;}
    void setPrevious(DNode *previous) { d_previous = previous;}
    void setNext(DNode *next) { d_next = next;}
    string data() const { return d_data;}
    DNode *next() const { return d_next;}
    DNode *previous() const { return d_previous;}
    void print() const { cout << d_data << " ";}
};
```

Listas Duplamente Ligadas

- **Travessia – O(n):**
  - percorrer todos elementos da lista
- **Algoritmos (em frente e para trás):**

```
Node *aux = first;
While (aux!=0)
{
    // processamento
    aux = aux->next();
}
```

- **Remover – procurar elemento, refazer ligações e apagar elemento**

Listas Duplamente Ligadas

```
class DList  // lista duplamente ligada de string
{
  private:
    DNode *first; // primeiro da lista
    DNode *last; // ultimo da lista
  public:
    DList(){ first=0; last=0;} // lista ligada vazia
    ~DList()  // destrutor
    {
        DNode *aux = first; DNode *cur;
        while(aux!=0) { cur = aux; aux = aux->next(); delete cur;}
    }
    bool empty() const { return (first==0);} // O(1)
    void print() const // O(n)
    {
        DNode *aux=first;
        while(aux!=0) { aux->print(); aux = aux->next();}
        cout << "\n";
    }
```

Listas Duplamente Ligadas

```
int size() const
{
    DNode *aux = first; int i = 0;
    while(aux!=0) { i++; aux = aux->next();}
    return i;
}
DNode *find(string elem) // O(n)
{
    DNode *aux=first;
    while(aux!=0 && aux->data()!=elem) aux = aux->next();
    if(aux!=0) return aux;
    else { cerr << "Elem is not in list\n"; return 0;}
}
void insert_head(string elem) // O(1)
{
    DNode *node = new DNode(elem,0,first);
    if(last==0) last = node;
    if(first!=0) first->setPrevious(node);
    first = node;
}
```


Listas Duplamente Ligadas

```
void insert_end(string elem) // O(n)
{
    DNode *node, *aux;
    node = new DNode(elem,0,0);
    last = node; aux = first;
    if(aux==0) first = node;
    else {
        while(aux->next()!=0) aux = aux->next();
        aux->setNext(node);
        node->setPrevious(aux);
    }
}
void insert_before(string elem, DNode *B) // metodo auxiliar
{
    DNode *node = new DNode(elem,B->previous(),B);
    if(B->previous()==0) first = node; else B->previous()->setNext(node);
    B->setPrevious(node);
    if(B->next()==0) last = B;
}
```

Listas Duplamente Ligadas

```
void insert_sort(string elem)
{
   DNode *node, *aux;
   aux = first;
   while(aux!=0 && aux->data()<elem) aux = aux->next();
   if(aux!=0) insert_before(elem,aux); else insert_end(elem);
}
DNode *remove(string elem)
{
   DNode *aux = first;
   while(aux!=0 && aux->data()!=elem) aux = aux->next();
   if(aux==0) cerr << "Item not found\n";
   else {
       if(aux->previous()!=0) (aux->previous())->setNext(aux->next());
       else first=aux->next();
       if(aux->next()!=0) (aux->next())->setPrevious(aux->previous());
       else last=aux->previous();
       delete aux;
   }
}
}; // fim da classe
```

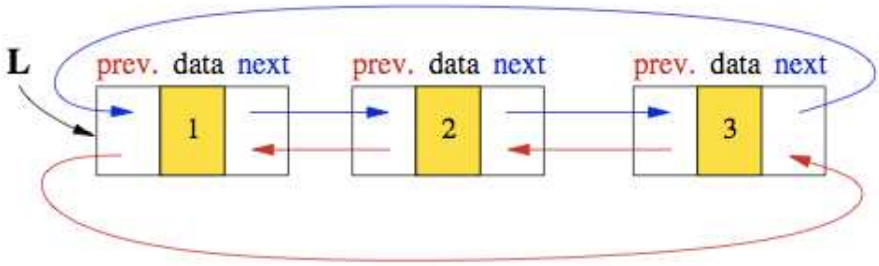

Listas Duplamente Ligadas

```
int main()
{
    DList DL;
    DL.insert_sort("d");
    DL.insert_sort("a");
    DL.insert_sort("c");
    DL.insert_sort("b");
    DL.insert_sort("e");
    DL.print();
    DL.print_reverse();
    DL.remove("b");
    DL.remove("d");
    DL.print();
    DL.remove("a");
    DL.remove("g");
    DL.print();
    return 0;
}
```

O Que aparece no écran?

Listas Circulares

- **O último nodo liga-se ao primeiro, deixa de haver ligações 0 (NULL)**
- **Permite uma navegação completa a partir de qualquer nodo**
- **Pode haver listas circulares simples e duplamente ligadas**



Implementação (no exemplo):

- Existem 2 classes: Node e ListC;
- Node: contém o atributo d_data, d_next e d_data (será do tipo int, mas pode ser uma qualquer classe…
- List: contém os atributos first e last, do tipo Node* e que apontam para o primeiro e último nodo da lista…

Listas Circulares

```
class ListC // lista ligada de int,
{
  private:
    Node *first; Node *last;
  public:
    ListC(){ first = 0; last = 0;} // lista ligada vazia
    ~ListC() { // destrutor
        Node *aux=first; Node *cur;
        do { cur = aux; aux = aux->next(); delete cur;}
        while (aux!=first);
    }
    bool empty() const { return (first==0);} // O(1)
    void print() const { // O(n)
        Node *aux = first;
        do { aux->print(); aux = aux->next();}
        while (aux!=first);
        cout << "\n";
    }
```

Listas Circulares

```
void print(Node *N) const { // O(n)
   Node *aux=N;
   do { aux->print(); aux = aux->next(); } while (aux!=N);
   cout << endl;
}
Node *find(int elem) // O(n)
{
   Node *aux = first;
   if(aux!=0) {
       do {
           if(aux->data()==elem) return aux; else aux = aux->next();
       } while(aux!=first);
   }
   cerr << "Elem is not in list\n"; return 0;
}
void insert_head(int elem) {  // O(1)
   Node *node = new Node(elem,first);
   if(last==0) last = node;
   first = node; last->setNext(first);
}
}; // fim da classe
```


Listas Circulares

```
int main()
{
    ListC LC;
    LC.insert_head(3);
    LC.insert_head(2);
    LC.insert_head(1);
    LC.print();
    Node *N = LC.find(2);
    LC.print(N);
    N = LC.find(3);
    LC.print(N);
    return 0;
}
```

O Que aparece no écran?

Eficiência das Estruturas de Dados

	Acesso	Comentário
Pilha	Apenas ao elemento mais recente O(1)	Muito rápido
Fila	Apenas ao elemento menos recente O(1)	Muito rápido
Lista Ligada	Qualquer item O(N)	
Árvore de Pesquisa	Qualquer item por nome ou ordem O(log N)	Caso médio; em árvores especiais é pior caso
Tabela de Dispersão	Qualquer item por nome O(1)	Quase garantido
Fila de Prioridade	Acesso ao mínimo: O(1) Apagar mínimo: O(log N)	Inserção: O(1) caso médio, O(log N) pior caso

Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 10

Estuturas de Dados: Listas, Pilhas e Filas

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em Cortez 2011 , Reis, Rocha e Faria, 2007)