Universidade de Brasília

Listas Ligadas

Tópico 2 - Algoritmos e Estruturas de Dados

Prof. Daniel Guerreiro e Silva

Roteiro

1. Listas ligadas
2. Análise de complexidade de algoritmos
3. Listas duplamente ligadas

Leitura recomendada para o tópico

- Adam Drozdek - Estruturas de Dados e Algoritmos em C++ - Tradução da 4ª edição americana
  - Capítulo 2
  - Capítulo 3, seções 3.1, 3.2, 3.7-3.9
- http://visualgo.net
- Thomas H. Cormen *et al.* - Algoritmos: Teoria e Prática 3ª edição (Leitura complementar)
  - Seções 10.2 e 10.3

Listas ligadas

Estruturas de Dados

- A programação está constantemente envolvida com a manipulação de **conjuntos de dados**
- Esses conjuntos são **dinâmicos**, isto é, podem crescer, diminuir, alterar os seus elementos, etc...
- O elemento de um conjunto é representado por um objeto com atributos que podem ser examinados e manipulados
  - Um dos atributos pode ser uma **chave de identificação**
  - O objeto pode conter **dados satélite** também

Estruturas de Dados

- Dependendo do algoritmo a ser implementado, diferentes **operações** nos conjuntos podem ser exigidas
  1. Busca,
  2. Inserção,
  3. Remoção,
  4. Mínimo,
  5. Máximo,
  6. Sucessor,
  7. Antecessor.

Estruturas de Dados

- Portanto, a escolha adequada de uma estrutura de dados (conjunto dinâmico) para o algoritmo depende das operações que devem ser suportadas e do tempo que demoram para executar
- Este é o contexto para se estudar estruturas de dados e suas aplicações
  - Listas
  - Pilhas, Filas
  - Árvores
  - Grafos
  - Dicionários

Matrizes e vetores

- Muito úteis, são as primeiras estruturas de dados a se estudar, geralmente, em programação
- Porém, elas possuem desvantagens:

1. O tamanho deve ser conhecido no momento da compilação
2. Os dados na memória são armazenados de forma contígua, o que complica a inserção de um elemento dentro dela

Listas Ligadas

- Listas ligadas são uma estrutura linear formada por uma coleção de **nós**
- Cada nó é um objeto com o **atributo chave** e um **ponteiro** (endereço) para o próximo nó
- Dessa forma, cada nó pode estar em regiões diferentes da memória

Nó de uma lista ligada de inteiros em C++

```
class Node{
  public:
     int data; //chave
     Node *next; //aponta o proximo nó

     //construtores
     Node();
     Node(int, Node*);
};
```

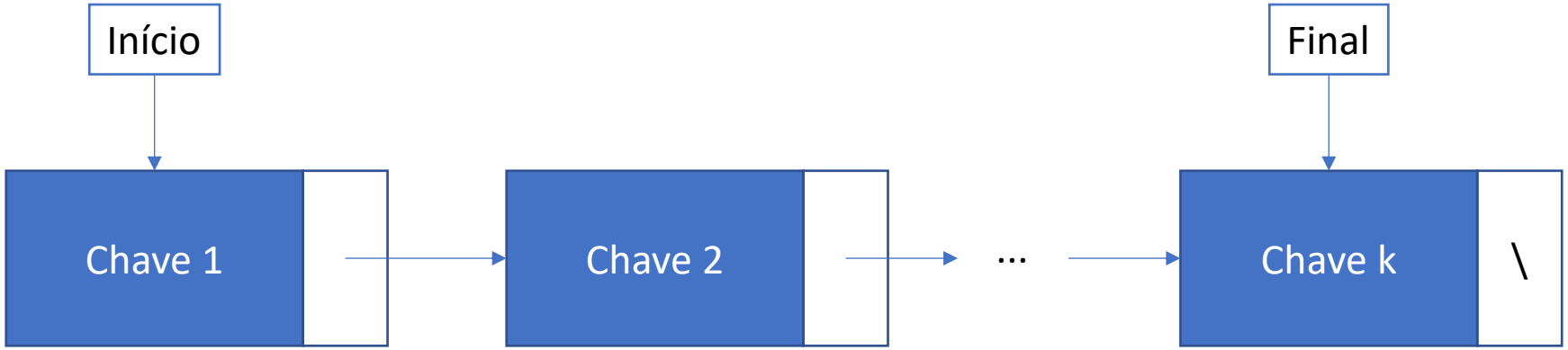
Nó de uma lista ligada de inteiros em C++

```
Node::Node(){
   next = 0;
}

Node::Node(int i, Node *prox){
   data = i;
   next = prox;
}
```

Listas Ligadas

- Em termos práticos, a implementação da lista ligada contém um apontador para o primeiro elemento (início ou ”cabeça” da lista) e outro para o final.



Operações em Listas Ligadas

1. Inserção de um nó no início da lista
2. Inserção de um nó no fim da lista
3. Remoção de um nó no início da lista
4. Remoção de um nó no fim da lista
5. Busca de um nó pela chave

Classe linkedList em C++

```
class linkedList{
public:
  linkedList();  //construtor
  ~linkedList(); //destrutor
  //metodos
  bool isEmpty(); //verifica se esta vazia
  bool inList(int); //busca inteiro na lista e retorna true se positivo
  void addHead(int); //adiciona inteiro ao inicio da lista
  void addTail(int); //adiciona inteiro ao final da lista
  int deleteHead(); //remove primeiro elemento da lista e o retorna
  int deleteTail(); //remove ultimo elemento da lista e o retorna
private:
  Node *head; //ponteiro para o inicio da lista ligada
  Node *tail; //ponteiro para o final da lista ligada
};
```

Observação: destrutores em C++

- Destrutores são funções invocadas quando um objeto está para ``morrer''.
- Caso um objeto tenha recursos alocados, destrutores devem liberar tais recursos.
- Por exemplo, se o construtor de uma classe alocou uma variável dinamicamente com **new**, o destrutor correspondente deve liberar o espaço ocupado por esta variável com o operador **delete**.
- A função destrutor de uma classe tem o mesmo nome da classe com um til (~) como prefixo.
  - Ex.: um destrutor para a classe *rectangle* seria chamado ~*rectangle()*.

Relembrando: alocação de memória em C++

- A alocação de memória permite criar, **dinamicamente**, novas áreas de dados e referenciá-las através de um ponteiro, i.e. uma variável que armazena o endereço para esta área
- Em C++ isto é feito com o comando **new**

```
double *p = new double;
int *q = new int;
```

Relembrando: alocação de memória em C++

- Pode-se chamar **new** para alocar memória e já inicializar um objeto com argumentos para o construtor:

```
Classe *r = new Classe(<arg. construtor>);
```

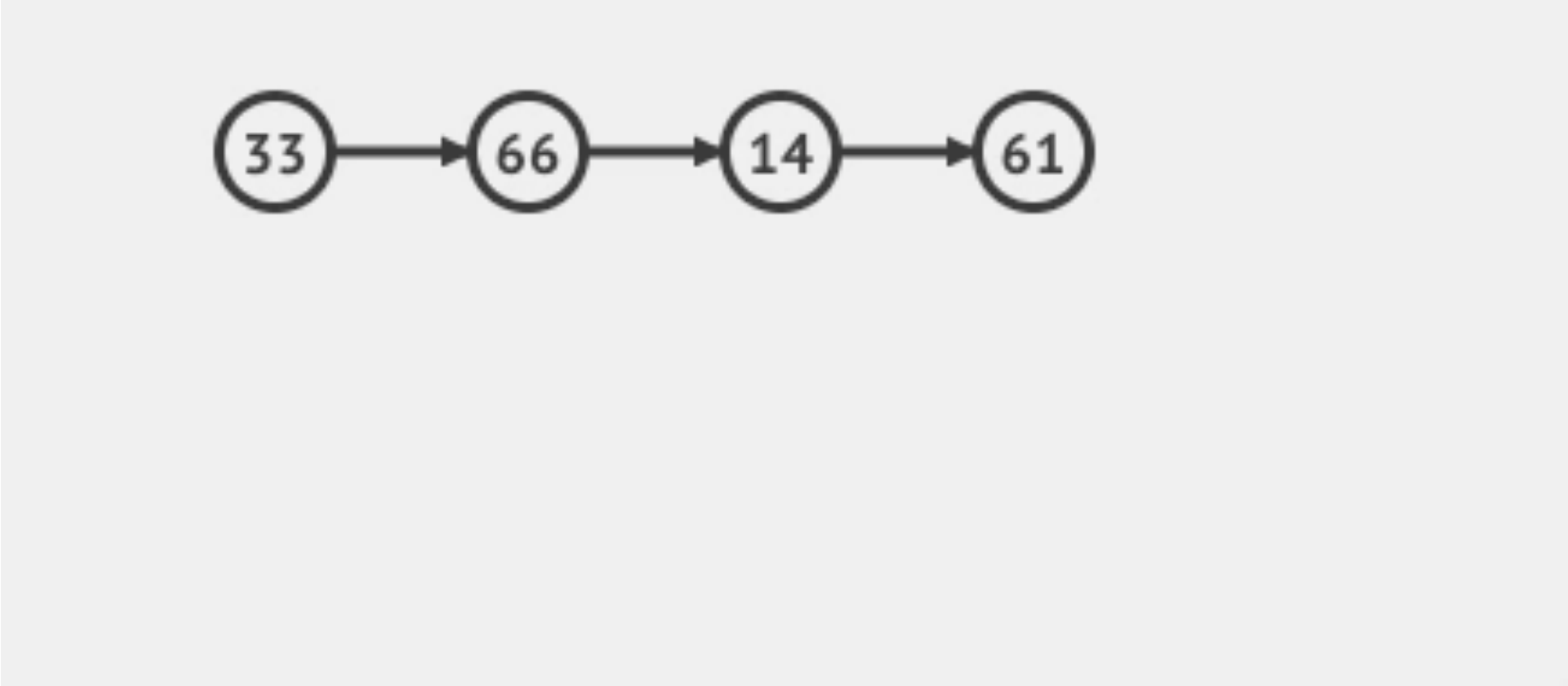
- Quando terminar de se usar uma área de memória alocada, é importante realizar o procedimento de desalocação através do comando **delete**:

```
delete r;
```

Inserção no início da linkedList

1. Crie um novo nó e atribua a chave a ele (membro `data`)
2. O membro `next` deve apontar para o primeiro elemento da lista, i.e. `head`
3. Atualize `head` para apontar para o novo nó

Inserção no início da linkedList



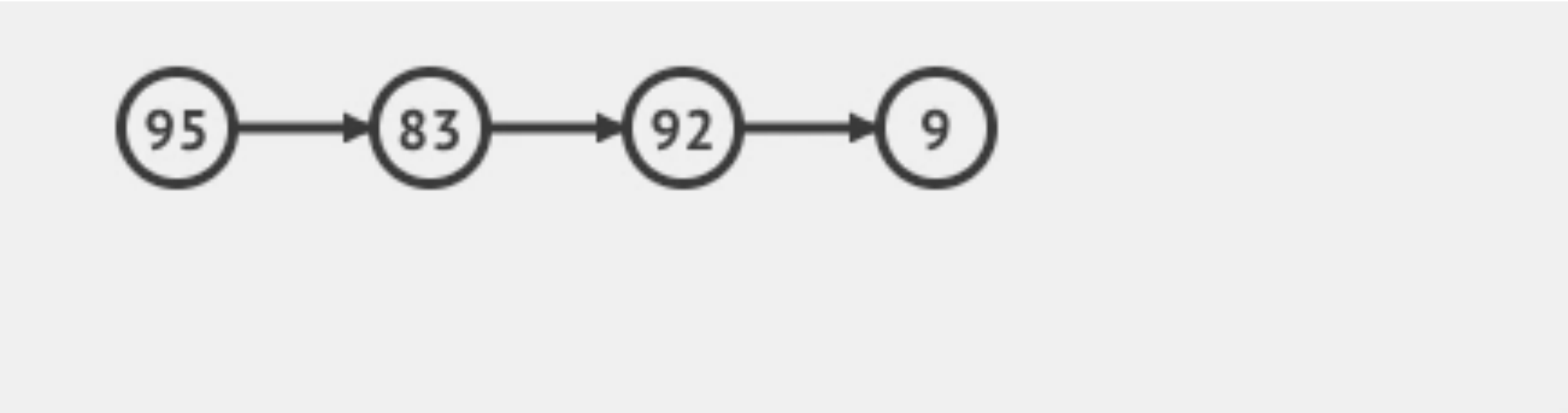
Inserção no início da linkedList

```
void linkedList::addHead(int info)
{
  head = new Node(info, head);
  if (tail == 0) //se lista vazia
    tail = head;
}
```

Inserção no final da linkedList

1. Crie um novo nó e atribua a chave a ele (membro `data`)
2. O seu membro `next` deve apontar para nulo/vazio, pois será o último elemento
3. Inclua o nó na lista: faça o membro next do último elemento apontar para o recém criado
4. Atualize `tail` para apontar para o novo nó

Inserção no final da linkedList



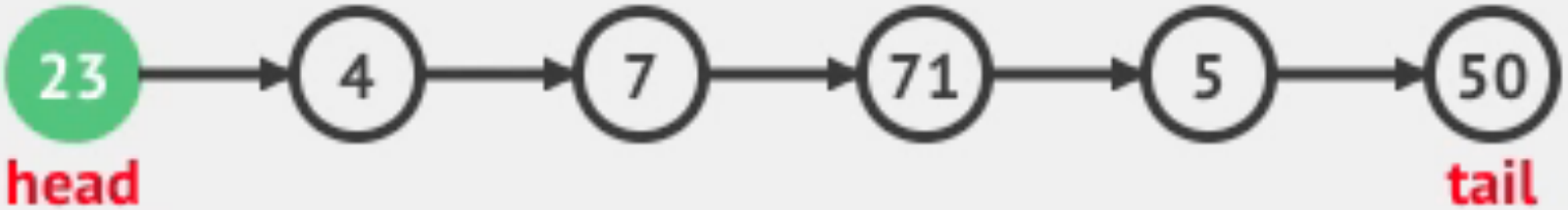
Inserção no final da linkedList

```
void linkedList::addTail(int info){
  if (tail != 0) //se a lista nao esta vazia
  {
    tail->next = new Node(info, 0);
    tail = tail->next;
  }
  else
  {
    tail = new Node(info,0);
    head = tail; //apontam para o único elemento
  }
}
```

Remoção no início da linkedList

1. Armazene o dado do nó inicial em um espaço temporário
2. Atualize `head` para apontar para o segundo nó da lista, i.e. o membro `head->next`
3. Destrua (desaloque) o nó inicial e retorne o dado

Remoção no início da linkedList



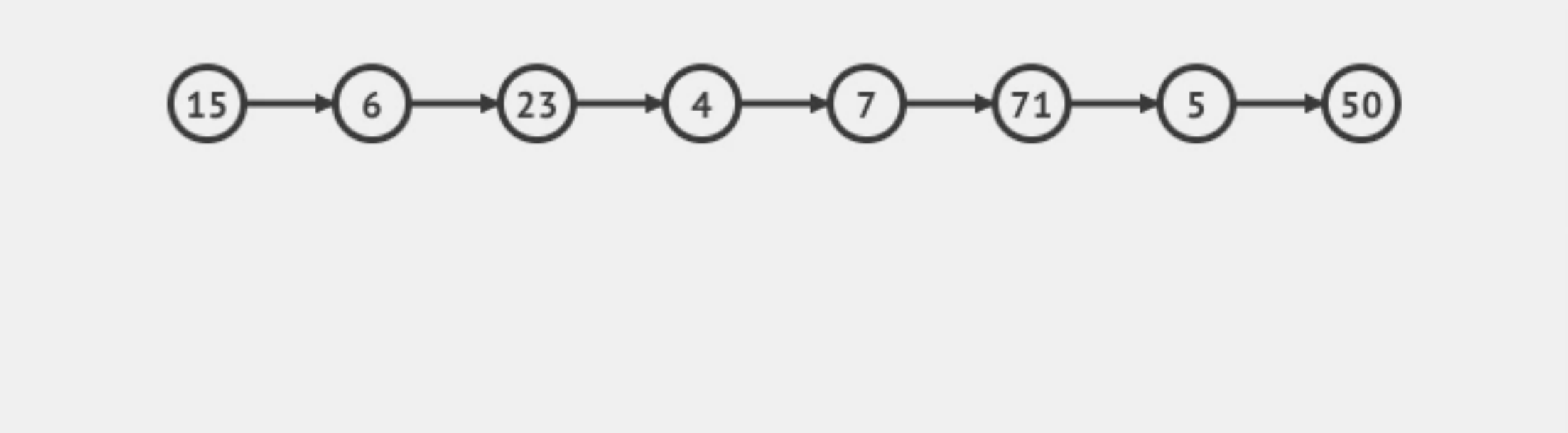
Remoção no início da linkedList

```
int linkedList::deleteHead(){
  int retvalue = head->data;
  Node *temp = head;
  if(head == tail){ //há um elemento só
    head = 0;
    tail = head;
  }
  else
    head = head->next;
  delete temp; //destroi o nó antigo
  return retvalue;
}
```

Remoção no final da linkedList

1. Procure o antecessor do último elemento da lista
2. Apague o último elemento
3. Apontador `tail` deve apontar agora para o antecessor
4. Novo último elemento deve apontar para vazio, i.e. membro `next` aponta para vazio

Remoção no final da linkedList



Remoção no final da linkedList

```
int linkedList::deleteTail(){
  int retvalue = tail->data;
  if (tail == head){ //só há um elemento na lista
    delete head;
    head = tail = 0;
  }
  else{
    Node *temp;
    //procura o antecessor do ultimo elemento da lista
    for(temp = head; temp->next != tail; temp = temp->next );
    delete tail;
    tail = temp; //antecessor passa a ser o ultimo da lista
    tail->next = 0;
  }
  return retvalue;
}
```

Busca pela chave k

1. Deve-se, partindo do apontador para o início da lista, percorrer nó a nó checando se a chave é igual a k
2. Se chegar ao final da lista (apontador para o próximo nó é 0/vazio/NULL), significa que k não está presente

Busca pela chave k

```
bool linkedList::inList(int key)
{
  Node *i;
  //caminha com apontador i ate o fim da lista
  //OU ate encontrar key
  for (i = head; i != 0 && i->data !=  key; i = i->next);
  return (i != 0);
}
```

Implementação completa em C++

- Estude, compile e execute a implementação completa da lista ligada de inteiros em C++
  - Arquivos list.hpp, list.cpp e list_main.cpp
  - Sugestão: aprenda a criar arquivos makefile e usar o comando make em http://www.klebermota.eti.br/2013/03/11/usando-o-gcc-e-o-make-para-compilar-lincar-e-criar-aplicacoes-cc/ http://stackoverflow.com/questions/2481269/how-to-make-a-simple-c-makefile

Exercício de Assimilação de Conceitos

1. Atualize a biblioteca de lista ligada de inteiros (arquivos list.h e list.cpp) para que contenham as novas funções-membro:
   ```
   int linkedList::max();
   double linkedList::avg();
   ```
2. Faça um programa que leia inteiros da entrada e guarde-os na lista ligada, até o usuário digitar o sinal EOF (CTRL-D no Linux/Mac ou CTRL-Z no Windows). Então o programa deve mostrar o valor máximo e a média.

Análise de complexidade de algoritmos

Análise de Algoritmos

- Além de aprender novos algoritmos, é importante saber analisar o desempenho deles.
- Isto significa, principalmente, avaliar o **tempo de execução** de cada um
- Mas em que máquina?
  - Modelo de máquina teórica, com uma só CPU, que possui as operações aritméticas (soma, subtração, produto, divisão...), de controle (condicional, subrotinas) e de movimentação de dados (leitura, escrita, cópia).

Análise de Algoritmos

- O tempo que um algoritmo leva para se encerrar geralmente cresce com o tamanho da entrada fornecida
- Logo, costuma-se descrever o tempo de execução de um programa em função do tamanho da sua entrada
- O tempo de execução para uma dada entrada é o número de operações primitivas realizadas pelo algoritmo

Notação big-O

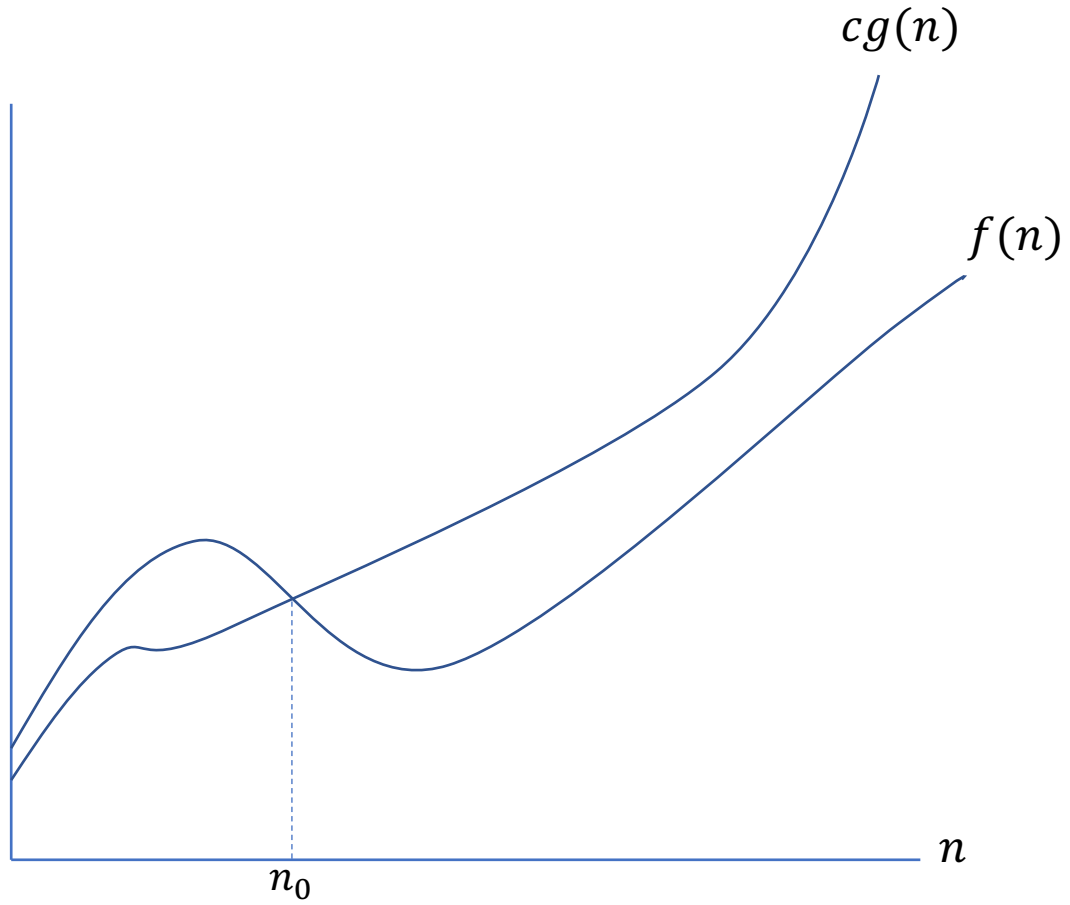
Para uma dada função $g(n)$, denotamos por $O(g(n))$ o conjunto de funções

$$O\big(g(n)\big) = \{f(n): existem\ constantes\ positivas\ c\ e\ n_0$$
$$tais\ que\ 0 \leq f(n) \leq cg(n)\ para\ todo\ n \geq n_0\}$$

- A notação big-O define um **limite assintótico superior** para funções

Notação big-O

$f(n) = O(g(n))$



Análise de Algoritmos

- Para fins de generalização, analisa-se o tempo de execução **assintótico** dos algoritmos
  - O que importa não é o tempo, em si, mas sim a sua **taxa de crescimento**
- Logo, fatores constantes e termos de menor ordem podem ser descartados, no cálculo final
- Vejamos como isso se dá, na prática: analisemos, em termos de operações de comparação e de atribuição, a inserção, remoção e busca em listas singularmente ligadas

Análise - Inserção no início da lista

Custo em função do tamanho da lista (n)

```
void linkedList::addHead(int info)
{
  head = new Node(info, head);        C1
  if (tail == 0) //se lista vazia     C2
    tail = head;                      C1
}
```

Custo total

$$T(n) = 2c_1 + c_2$$

Análise – Inserção no início da lista

- Pela notação big-O, podemos concluir que

$$T(n) = O(1)$$

afinal, existe constante c e n_0 tal que

$$c \cdot 1 \geq T(n) = 2c_1 + c_2, n \geq n_0$$

Análise - Inserção no final da lista

Custo em função do tamanho da lista (n)

```
void linkedList::addTail(int info){
  if (tail != 0){                          C1
    tail->next = new Node(info, 0);        C2
    tail = tail->next;                     C2
  }
  else{
    tail = new Node(info,0);               C2
    head = tail;                           C2
  }
}
```

Custo total

$$T(n) = O(1)$$

Análise – Remoção no início da lista

```
int linkedList::deleteHead(){
  int retvalue = head->data;
  Node *temp = head;
  if(head == tail)
    head = tail = 0;
  else
    head = head->next;
  delete temp;
  return retvalue;
}
```

Custo em função do tamanho da lista (n)

C_1

C_1

C_2

$2*C_1$

C_1

Custo total

$$T(n) = O(1)$$

Análise – Remoção no final da lista

Custo em função do tamanho da lista (n)

```
int linkedList::deleteTail(){                                              
  int retvalue = tail->data;                                               C1
  if (tail == head){ //só há um elemento na lista                          C2
    delete head;
    head = tail = 0;                                                       2*C1
  }
  else{
    Node *temp;
    for(temp = head; temp->next != tail; temp = temp->next );              (n-1).C1 + (n-1)C2
    delete tail;
    tail = temp;                                                           C1
    tail->next = 0;                                                        C1
  }
  return retvalue;
}
```

Custo total

$$T(n) = O(n)$$

Análise da Busca, pior caso: chave não está na lista

Custo em função do tamanho da lista (n)

```
bool linkedList::inList(int key)
{
  Node *i;
  //caminha com apontador i ate o fim da lista
  //OU ate encontrar key
  for (i = head; i != 0 && i->data !=  key; i = i->next);   n.C1 + n.C2
  return (i != 0);                                          C2
}
```

Custo pior caso

$$T(n) = O(n)$$

Análise da Busca, melhor caso: chave é a primeira da lista

Custo em função do tamanho da lista (n)

```
bool linkedList::inList(int key)
{
  Node *i;
  //caminha com apontador i ate o fim da lista
  //OU ate encontrar key
  for (i = head; i != 0 && i->data !=  key; i = i->next);    C1 + 2.C2
  return (i != 0);                                           C2
}
```

Custo melhor caso

$$T(n) = O(1)$$

Análise da busca, caso médio

- Assuma que qualquer nó da lista tem a mesma chance de ser buscado, para uma longa sequência de buscas, teremos um número médio de operações de comparação igual a

$$\frac{1 + 2 + \cdots + n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n+1}{2}$$

Portanto, na média a busca por uma chave k leva $O(n)$ operações para encerrar

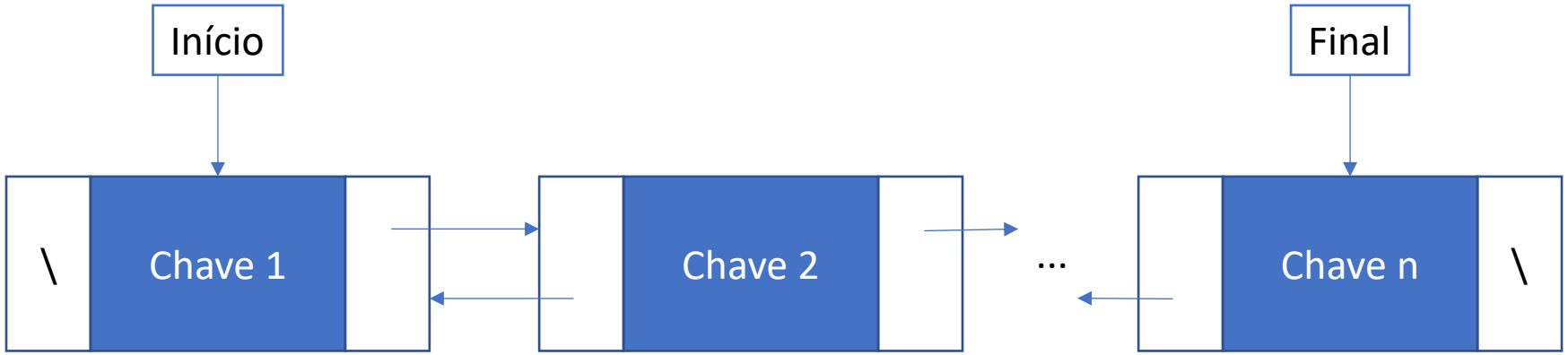
Listas duplamente ligadas

Listas duplamente ligadas

- Note que a lista singularmente ligada implica numa operação de remoção no final da lista com custo $O(n)$
- Para listas longas ou com muitas operações de remoção, isto pode ser uma barreira ao processamento rápido da lista
- Por isso podemos utilizar uma variação conhecida como **lista duplamente ligada**

Listas duplamente ligadas

- Cada nó é um objeto com o **atributo chave** e dois ponteiros: um para o sucessor e outro para o antecessor



Listas duplamente ligadas

- A implementação de uma lista duplamente ligada é bem similar à das listas singularmente ligadas, salvo algumas adaptações
- Destaca-se neste caso a nova operação de remoção de um elemento do final, que agora terá custo computacional $O(1)$.

Classe doubleNode para inteiros

```
class doubleNode
{
public:
   int data; //chave
   doubleNode *previous, *next;
   //construtores
   doubleNode();
   doubleNode(int, doubleNode*, doubleNode*);
};
```

Definições – classe doubleNode

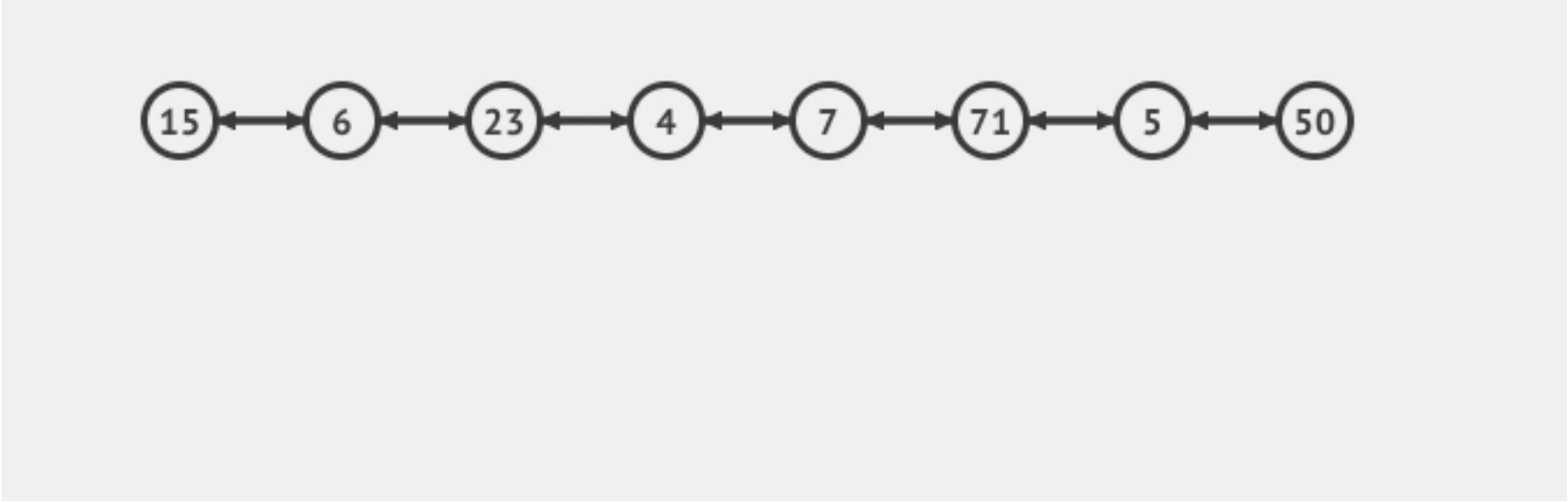
```
doubleNode::doubleNode(){
  previous = 0;
  next = 0;
}

doubleNode::doubleNode(int i, doubleNode *prev = 0, doubleNode *prox = 0){
  data = i;
  previous = prev;
  next = prox;
}
```

Classe doubleLinkedList

```
class doubleLinkedList
{
public:
  doubleLinkedList();  //construtor
  ~doubleLinkedList(); //destrutor
  //metodos
  bool isEmpty();
  bool inList(int);
  void addHead(int);
  void addTail(int);
  int deleteHead();
  int deleteTail();
private:
  doubleNode *head; //inicio da lista duplamente ligada
  doubleNode *tail; //final da lista duplamente ligada
};
```

A nova operação de remoção do final da lista



1. Armazene um apontador `tmp` para o último elemento
2. Final da lista deve apontar para o antecessor, i.e. `tail=tail->previous`
3. Último elemento deve apontar para vazio, i.e. `tail->next=0`
4. Apague elemento apontado por `tmp`

Implementação – remoção no final

```
int doubleLinkedList::deleteTail(){
  int retvalue = tail->data;
  if (tail == head){ //só há um elemento na lista
    delete head;
    head = 0;
    tail = head;
  }
  else{
    tail = tail->previous;
    delete tail->next; //apaga
    tail->next = 0;//novo final nao tem sucessor
  }
  return retvalue;
}
```

Análise – Remoção no final da lista duplamente ligada

```
int doubleLinkedList::deleteTail(){
  int retvalue = tail->data;
  if (tail == head){
    delete head;
    head = tail = 0;
  }
  else{
    tail = tail->previous;
    delete tail->next; //apaga
    tail->next = 0;
  }
  return retvalue;
}
```

Custo em função do tamanho da lista (n)

C_1

C_2

$2*C_1$

C_1

C_1

Custo total

$$T(n) = O(1)$$

Implementação - Lista duplamente ligada

- Estude, compile e execute a implementação completa nos arquivos
  - doublelist.h
  - doublelist.cpp
  - doublelist_main.cpp

A classe std::list da biblioteca padrão do C++

- A STL oferece um contêiner que implementa uma lista duplamente encadeada de tipo genérico
- Para usá-la deve-se colocar a diretiva no código
  `#include <list>`
- Para gerar uma nova lista basta declarar uma variável
  `std::list<T> lista;`
  onde T é o tipo do objeto a ser armazenado nela.

Algumas funções-membro da classe std::list

`iterator begin();`	Retorna iterador p/ começo da lista
`iterator end();`	Retorna iterador p/ final da lista
`void push_front (const value_type& val);`	Insere elemento no início
`void pop_front();`	Apaga elemento do início
`void push_back (const value_type& val);`	Insere elemento no final
`void pop_back();`	Apaga elemento do final

Veja outras funções em http://www.cplusplus.com/reference/list/list/

Estudo de caso: vetor vs. lista ligada

Problema

Um programa calcula a nota final de estudantes de um curso, com base nas notas das avaliações dadas na entrada. Uma lista de estudantes e suas notas finais é gerada de saída.

Deseja-se quebrar a lista em outras duas: uma com os estudantes aprovados, outra com os estudantes reprovados.

Qual estrutura é melhor usar, `vector` ou `list`?

Vetor vs. lista ligada

Descubramos a resposta na prática, com a implementação do programa em C++

- Baixe o arquivo aed_student.zip, na página do curso na plataforma Aprender 3, e estude a implementação
  - Para compilar invoque o comando `make` no diretório
- Compare a execução dos programas failsvetor e failslista com os arquivos de entrada notas100.txt e notas1000.txt

Vetor (Array) vs. Lista ligada

Vetor (array)	Lista ligada
Armazenamento contíguo na memória	Armazenamento esparso
Otimizado para acesso aleatório rápido	Otimizado para inserção e remoção rápida
Bom para crescimento / redução de dados no final da estrutura	Bom para crescimento / redução de dados em posições arbitrárias da estrutura