ESTRUTURA DE DADOS

UNIVESP Universidade Virtual do Estado de São Paulo

SEMANA 1

- Compreender a sintaxe da linguagem C++ para as construções básicas: comandos condicionais, comandos de repetição e declaração de funções;
- Compreender o uso das variáveis do tipo ponteiro;
- Compreender a organização dos dados em formato de vetores.

Especificam conceitualmente os dados, de forma a refletir um relacionamento lógico entre os dados e o domínio de problema considerado. Além disso, as estruturas de dados incluem operações para manipulação dos seus dados, que também desempenham o papel de caracterização do domínio de problema considerado.

Dados Estruturados: permitem agregar mais do que um valor em uma variável, existindo uma relação estrutural entre seus elementos. Os principais são arranjos (também denominados vetores e matrizes, utilizados para agregar componentes do mesmo tipo, com um tamanho máximo predefinido), registros (para agregar componentes de tipos diferentes),seqüências (para coleções ordenadas de componentes do mesmo tipo) e conjuntos (para definir, para um tipo básico, uma faixa de valores que seus componentes podem assumir), entre outros.

Tipos de Dados são diferentes de Estrutura de Dados

Enquanto **tipo** consiste da definição do conjunto de valores (denominado domínio) que uma variável pode assumir ao longo da execução de um pro- grama e do conjunto de operações que podem ser aplicadas sobre ele (como inteiros, lógicos, etc), **estrutura** permitem agregar mais do que um va- lor em uma variável, existindo uma relação estrutural entre seus elementos.

Principais Dados Estruturados

Arranjos: vetores e matrizes, utiliza- dos para agregar componentes do mesmo tipo, com um tamanho máximo predefinido.

Registros: para agregar componentes de tipos diferentes.

Sequências: para coleções ordenadas de componentes do mesmo tipo**.**

Conjuntos: para definir, para um tipo básico, uma faixa de valores que seus componentes podem assumir,

LINGUAGEM C++

Compilada, imperativa e de uso geral, com suporte a orientação a objetos.

Permite: a manipulação de ponteiros de maneira explícita, é orientada a objetos e permite a separação entre a visão lógica das estruturas e a implementação e deixa a cargo do programador as operações para gerenciamento das estruturas de dados.

Gera programas mais rápidos.
Linguagem Compilada.

Tipagem Estática.
Manipulação Explícita da Memória

Ponteiros: uma variável cujo conteúdo é um **endereço** de memória (localizado na memória de uma variável ou função), não um valor no sentido tradicional.
Conhecer o endereço de uma variável permite criar estruturas complexas e trabalhar diretamente com a memória permite a criação de programas mais eficientes.

START

Para inicializar um ponteiro: tipo *ponteiro;
O endereço de memória pode ser obtido utilizando o operador &:

```
int alpha;
int *intPointer;
intPointer = &alpha;
```

Existe a opção de alocação dinâmica pelo qual um programa aloca e libera memória em tempo de execução, eliminando a necessidade de definir o tamanho da memória, podendo alterá-la em tempo de execução. Para alocar ou desalocar na memória, os comanos new e delete são utiizados.

```
int *intPointer;
intPointer = new int;
```

Vetores: ***a maneira <u>mais simples</u> de estruturarmos um conjunto de dados***

Elementos devem ser do mesmo tipo;
Tamanho é fixado na declaração do vetor;
Ocupam tamanhos consecutivos de memória;

Exemplo: int c[10] = {14,0,3};

14	0	3	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

SINTAXE BÁSICA

```
int number1;
int number2;

std::cout << "Digite o primeiro número: ";
std::cin >> number1;
std::cout << "Digite o segundo número: ";
std::cin >> number2;

int   sum  = number1 + number2;
int   sub  = number1 - number2;
int   mul  = number1 * number2;
int   div  = number1 / number2;
float fdiv = (float)number1 / (float)number2;
int   res  = number1 % number2;
```

```
int number1;
int number2;

cout << "Digite o primeiro número: ";
std::cin >> number1;
cout << "Digite o segundo número: ";
std::cin >> number2;

if (number1 == number2)
  cout << number1 << " ==  " <<  number2 << std::endl;
if (number1 != number2)
  cout << number1 << " !=  " <<  number2 << std::endl;
if (number1 < number2)
  cout << number1 << " <  " <<  number2 << std::endl;
if (number1 > number2)
  cout << number1 << " >  " <<  number2 << std::endl;
```

```
int sum(int number1, int number2){
  return number1 + number2;
}

int sub(int number1, int number2){
  return number1 - number2;
}

int mul(int number1, int number2){
  return number1 * number2;
}

int idiv(int number1, int number2){
  return number1 / number2;
}
```

```
int number1;
int counter = 0;
int amount  = 0;

while (counter < 10) {

  cout << "Digite um número (" << counter << ")" << endl;
  std::cin >> number1;

  if (number1 < 5) {
    amount++;
  }
  counter++;
}
```

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

void troca_por_valor(int a, int b){
        int temp;
        temp=a;
        a=b;
        b=temp;
}
int main(){
        int a=2, b=3;
        cout<<"Antes: a = "<<a<<" b = " << b<<endl;
        troca_por_valor(a,b);
        cout<<"Depois: a = "<<a<<" b = "<< b << endl;
        return 0;
}
```

```
codigos$ g++ ex8_parametro_por_valor.cpp -o valor
codigos$ ./valor
Antes: a = 2 b = 3
Depois: a = 2 b = 3
```

- Compreender o significado de tipos abstratos de dados;
- Distinguir entre as três visões de uma estrutura de dados;
- Compreender as três visões através da implementação das estruturas de dados filas e pilhas como vetores.

UNIVESP Universidade Virtual do Estado de São Paulo

SEMANA 2

TAD (Tipo Abstrato de Dados): estruturas de dados capazes de representar os tipos de dados que não foram previstos no núcleo das linguagens de programação e que, normalmente, são necessários para aplicações específicas. Essas estruturas são divididas em duas partes: os dados e as operações.

A característica essencial de um TAD é a separação entre conceito e implementação, ou seja, existe uma distinção entre a definição do tipo e a sua representação, e a implementação das operações.

Os TADs são geralmente implementados em linguagens de programação através do conceito de bibliotecas. A implementação das operações propriamente dita é feita em um arquivo separado, sendo que qualquer alteração na implementação dessas operações implica somente na compilação do módulo envolvido.Por fim, basta somente fazer a ligação da interface e da implementação com as aplicações do usuário para gerar um programa executável.

Representação física de dados

Consideramos somente a forma utilizada para implementar os relacionamentos entre os diferentes nodos

CONTIGUIDADE

A ordem entre os nodos da estrutura armazenada é definida implicitamente pela posição ocupada pelos nodos na memória. Assim, cada posição contígua na memória armazena o conjunto de informações correspondente a um nodo, que pode ser simples ou complexo, apresentando um ou vários campos.

⊕ Proteção de memória, transferência de dados, simplicidade, representação e acesso.

⊖ Compartilhamento de memória, previsão de espaço físico, complexidade, inserção e exclusão de componentes.

ENCADEAMENTO

O espaço necessário para a representação dos dados pode ser alocado à medida que se torne necessário, através de alocação dinâmica. Apresenta seus nodos alocados em posições aleatórias na memória.

⊕ Compartilhamento de memória, maleabilidade e facilidade para inserção e remoção de dados.

⊖ Transferência de dados, gerência de memória mais onerosa, menos intuitivo e acesso.

PROGRAMAÇÃO ORIENTADA A OBJETOS

POO

Classe: unidade básica que encapsula atributos estáticos e comportamento dinâmicos em uma caixa. Cria objetos (instâncias) e a comunicação com os objetos é feita pelo uso da interface pública do objeto.

⊕ **isolamento:** quando alterações não afetam todo o sistema, facilita a adição de novas funcionalidades e correção de problemas

Pilha é uma estrutura linear na qual inserções e remoções ocorrem no topo da pilha.

inserções e remoções **ocorrem em tempo constante.** Em outras palavras, independem do número de elementos na estrutura.

Fiila é uma estrutura linear na qual as inserções ocorrem no final e as exclusões ocorrem no início

Inserções ocorrem no **final** e **remoções** ocorrem no **início**

IMPLEMENTAÇÃO

```
#include "stack.h"
#include <iostream>
using namespace std;
```

1 CONSTRUTOR E DESTRUTOR

```
Stack::Stack()
{
  length = 0;
  structure = new ItemType[MAX_ITEMS];
}

Stack::~Stack()
{
  delete [] structure;
}
```

2 VERIFICAÇÃO DE CHEIO E VAZIO

```
bool  Stack::isEmpty() const
{
  return (length == 0);
}

bool Stack::isFull() const
{
  return (length == MAX_ITEMS);
}
```

3 INSERINDO ELEMENTOS

```
void Stack::push(ItemType item)
{
  if (!isFull()){
    structure[length] = item;
    length++;
  } else {
    throw "Stack is already full!";
  }
}
```

4 REMOVENDO ELEMENTOS

```
ItemType Stack::pop()
{
  if (!isEmpty()){
    ItemType aux = structure[length - 1];
    length--;
    return aux;
  } else {
    throw "Stack is empty!";
  }
}
```

IMPRININDO

```
void Stack::print() const
{
  cout << "Pilha = ";
  for (int i = 0; i < length; i++) {
    cout << structure[i];
  }
  cout << endl;
}
```

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

1 CONSTRUTOR E DESTRUTOR

```
Queue::Queue()
{
  front = 0;
  back  = 0;
  structure = new ItemType[MAX_ITEMS];
}

Queue::~Queue()
{
  delete [] structure;
}
```

2 VERIFICAÇÃO DE CHEIO E VAZIO

```
bool Queue::isEmpty() const
{
  return (front == back);
}

bool Queue::isFull() const
{
  return (back - front == MAX_ITEMS);
}
```

3 INSERINDO ELEMENTOS

```
void Queue::enqueue(ItemType item)
{
  if (!isFull()){
    structure[back % MAX_ITEMS] = item;
    back++;
  } else {
    throw "Queue is already full!";
  }
}
```

4 REMOVENDO ELEMENTOS

```
ItemType Queue::dequeue()
{
  if (!isEmpty()){
    front++;
    return structure[(front-1) % MAX_ITEMS];
  } else {
    throw "Queue is empty!";
  }
}
```

IMPRININDO

```
void Queue::print() const
{
  cout << "Fila = ";
  for (int i = front; i < back; i++) {
    cout << structure[i % MAX_ITEMS];
  }
  cout << endl;
}
```

- Definir critérios para decidir qual forma de organização de dados na memória é mais adequada para cada situação;
- Compreender o encadeamento das estruturas lineares na memória por meio de ponteiros;
- Entender a diferença na implementação de pilhas e filas com vetores e com listas encadeadas.

UNIVESP Universidade Virtual do Estado de São Paulo

SEMANA 3

LISTAS LINEARES Funciona em array ordenado

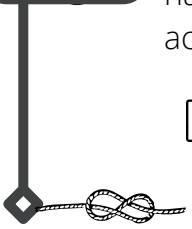
Cada elemento é precedido por um elemento e sucedido por outro, com exceção do primeiro que não tem predecessor e do último que não tem sucessor. Pilhas e filas são exemplos.
A ordem lógica dos elementos é igual a ordem física (Isto é, elementos vizinhos na lista estão em posições vizinhas de memória), o que confere acesso em tempo constante a qualquer tempo.

Precisamos alocar espaço suficiente para todos os elementos de uma só vez
Para manter a ordem, talvez sejam necessários muitos deslocamentos em memória.

LISTAS ENCADEADAS Não possui a vantagem de acesso constante a memória

Lista linear em que a ordem lógica dos elementos não é mesma da ordem física. Como é uma lista linear, cada elemento tem um sucessor e um predecessor.Os elementos estão espalhados na memória. Aqui a busca binaria perde sentido em vista da ausência do tempo constante de acesso ao meio da array.

Número de elementos pode aumentar ou diminuir durante a execução do programa.
A manutenção da ordem lógica não exigirá deslocamento de elementos.



```
#ifndef NODETYPE_H   // Inclua esse bloco apenas se TIME_H não está definido
#define NODETYPE_H   // Na primeira inclusão, defina TIME_H para que este bloco não seja incluído mais de uma vez.
typedef char ItemType;

struct NodeType
{
  ItemType info;
  NodeType* next;
};

#endif
```

```
#include "node_type.h"

class Stack
{
 public:
 Stack(); // Construtor
 ~Stack(); // Destrutor
 bool isEmpty() const;
 bool isFull() const;
 void print() const;

 void push(ItemType);
 ItemType pop();
 private:
 NodeType* structure;
};
```

```
#include "stack.h"
#include <cstddef> // Para funcionar o NULL
#include <new>

#include <iostream>
using namespace std;

Stack::Stack(){
  structure = NULL;
}

Stack::~Stack(){
  NodeType* tempPtr;
  while (structure != NULL) {
    tempPtr = structure;
    structure  = structure -> next;
    delete tempPtr;
  }
}

bool Stack::isEmpty() const {
  return (structure == NULL);
}

/*
  Aqui eu simplesmente verifico se o usuário possui memória
  disponível para alocar um novo elemento.
*/
bool Stack::isFull() const {
  NodeType* location;
  try {
    location = new NodeType;
    delete location;
    return false;
  } catch(std::bad_alloc exception){
    return true;
  }
}
```

```
/*
  Adicionar um novo item no topo da pilha. O nó que acabamos de criar
  deve apontar para quem estava no topo da pilha e o ponteiro
  structure deverá apontar para o novo nó.
 */
void Stack::push(ItemType item){
  if (!isFull()){
    NodeType* location;
    location = new NodeType;
    location->info = item;
    location->next = structure;
    structure = location;
  } else {
    throw "Stack is already full!";
  }
}

/*
  Devolve o objeto que está no topo da pilha.
 */
ItemType Stack::pop(){
  if (!isEmpty()) {
    NodeType* tempPtr;
    tempPtr = structure;
    ItemType item = structure->info;
    structure       = structure->next;
    delete tempPtr;
    return item;
  } else {
    throw "Stack is empty!";
  }
}

void Stack::print() const
{
  NodeType* tempPtr = structure;
  while (tempPtr != NULL) {
    cout << tempPtr->info;
    tempPtr  = tempPtr->next;
  }
  cout << endl;
```

UNIVESP
Universidade Virtual do Estado de São Paulo

SEMANA 4

- Entender os conceitos gerais de Tabela Hash;
- Compreender os fatores que influenciam na ocorrência de colisões;
- Compreender como o tratamento de colisões pode ser feito.

TABELA HASH: permitem buscas em tempo constante, satisfeitas algumas restrições

Essa estrutura pode ter vários nomes como: dicionários, mapas, arrays associativos, e assim por diante. Podemos utilizar as tabelas hash sempre que queremos armazenar uma coleção de dados para depois obter os registros de maneira eficiente

A função hash(chave) deve ser determinística. Para uma determinada chave a função sempre retorna o mesmo valor de hash. Por ser utilizada como uma função de indexação, a função de hash deve sempre retornar um valor de hash dentro dos limites da tabela [0,N−1], onde N é o tamanho da tabela. O método aproveita a possibilidade de acesso randômico à memória para alcançar uma complexidade média por operação de O(1), sendo o pior caso, entretanto, O(n).

- **Função de dispersão h:** Uma função de dispersão h transforma uma chave x em um endereço-base h(x) da tabela de dispersão.

Método da Divisão: A chave x é dividida pela dimensão da tabela m, e o resto da divisão é usado como endereço-base. **h(x) = h mod m**

Método da Dobra: A chave x é dividida pela dimensão da tabela m, e o resto da divisão é usado como endereço-base.

Método da Multiplicação: A chave é multiplicada por ela mesma (ou, uma alternativa, por uma constante), e o resultado é armazenado numa palavra de memória de b bits. O número de bits necessário para formar o endereço-base de uma chave é então retirado dos b bits, descartando-se os bits excessivos da extrema direita e da extrema esquerda da palavra.

Método da Análise dos Dígitos: De todos os métodos apresentados até agora, este é o único que leva em consideração o conhecimento prévio do tipo de chave que se busca. Ele é usado em geral para chaves decimais. A função de dispersão consiste em selecionar, de forma conveniente, alguns dos dígitos decimais que formam a chave para compor o seu endereço-base.

- **Colisão:** a função de dispersão pode não garantir injetividade, pois é possível a existência de outra chave $y \neq x$, tal que $h(y) = h(x)$. O compartimento h(x) já poderia então estar ocupado pela chave y. Em outras palavras: o mesmo endereço-base pode ser encontrado para chaves diferentes, como resultado da função de dispersão.

fator de carga: $\alpha = n/m$, onde n é o número de chaves armazenadas. Um método para diminuir colisões é então reduzir o fator de carga; à medida que este cresce, a possibilidade de ocorrerem colisões também cresce.

Uma ideia natural para tratar colisões consiste em armazenar as chaves sinônimas em listas encadeadas, há duas opções:

Encadeamento Exterior: consiste em manter m listas encadeadas, uma para cada possível endereço-base. Um campo para o encadeamento deve ser acrescentado a cada nó. Os nós correspondentes aos endereços-base serão apenas nós-cabeça para essas listas.

Encadeamento Interior: prevê a divisão da tabela T em duas zonas, uma de endereços-base, de tamanho p, e outra reservada aos sinônimos, de tamanho s. Naturalmente, p + s = m. Os valores p e s são fixos. Assim sendo, a função de dispersão deve obter endereços-base na faixa [0, p – 1] apenas. Nesse caso, $\alpha = n/m \leq 1$. A estrutura da tabela é a mesma que no caso do encadeamento exterior. Dois campos têm presença obrigatória em cada nó. O primeiro é reservado ao armazenamento da chave, enquanto o segundo contém um ponteiro que indica o próximo elemento da lista de sinônimos correspondentes ao endereço-base em questão.

Encadeamento Aberto: armazenar as chaves sinônimas também na tabela. Contudo, ao contrário do método de encadeamento interior, não há ponteiros. As chaves são armazenadas na tabela, sem qualquer informação adicional. Quando houver alguma colisão, determina-se, também por cálculo, qual o próximo compartimento a ser examinado. Se ocorrer nova colisão com alguma outra chave armazenada nesse último, um novo compartimento é escolhido mediante cálculo, e assim por diante. A busca com sucesso se encerra quando um compartimento for encontrado contendo a chave procurada. O indicativo de busca sem sucesso seria a computação de um compartimento vazio, ou a exaustão da tabela.

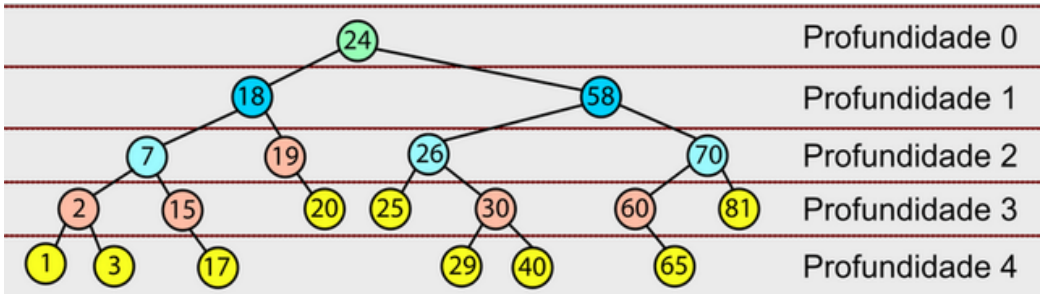
UNIVESP
Universidade Virtual do Estado de São Paulo

SEMANA 5

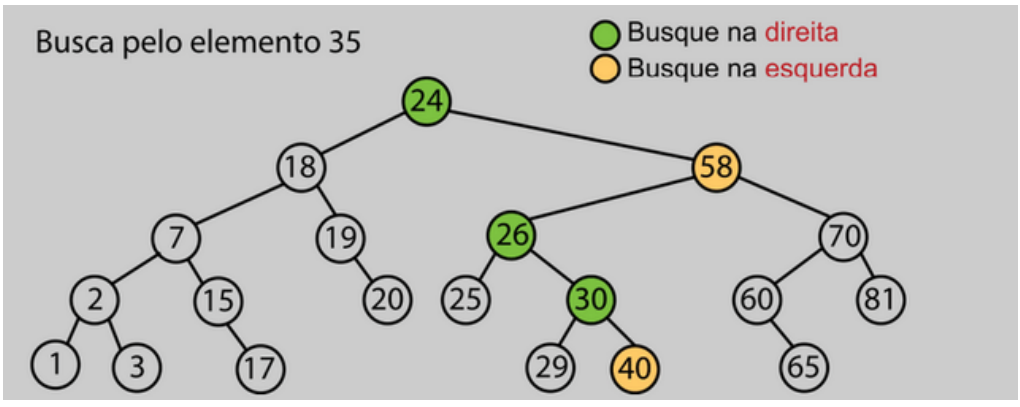
- Compreender a implementação e o uso de uma estrutura de dados não linear;
- Analisar as operações de inserção, remoção e busca em árvores binárias;
- Entender que as operações possuem tempo de processamento proporcional à altura da árvore.

ÁRVORES: é um conjunto de nós em que existe um nó raiz r, que contém zero ou mais subárvores cujas raízes são ligadas diretamente a r.
Uma subárvore também é uma árvore, não há um sucessor e um predecessor por nó, ou seja, não é uma estrutura linear que não são adequadas para representar hierarquia de dados.

Grau de um nó: número de subárvores desse nó.
Nó folha: nó de grau 0 (zero).
Nó interno: nó de grau maior que 0 (zero).
Descendentes: nós abaixo de um determinado nó
Altura de um nó: comprimento do caminho mais longo entre o nó até uma folha.
Altura da árvore: altura do nó raiz.
Profundidade de um nó: distância percorrida da raiz até o nó.
Árvore Binária: árvore em que abaixo de cada nó existem no máximo duas subárvores.

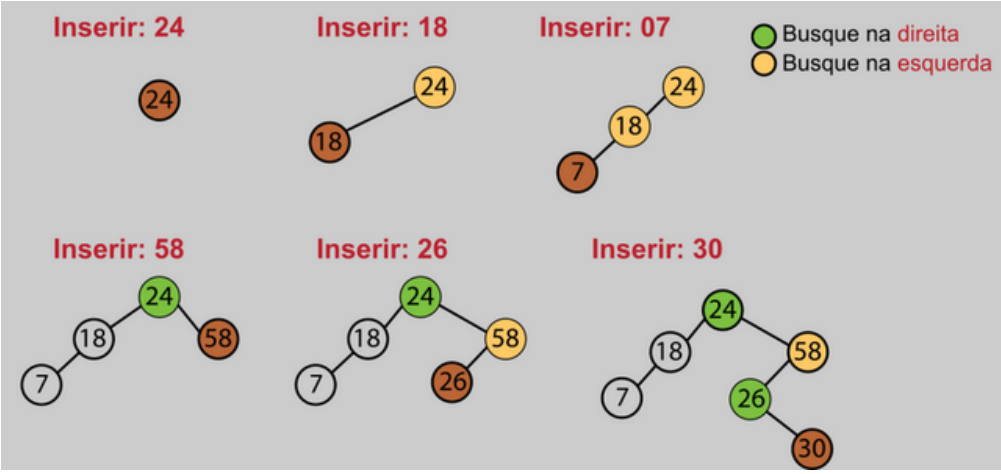


ÁRVORES BINÁRIAS DE BUSCA: a cada nó, todos os registros com chaves menores que a deste nó estão na subárvore da esquerda, enquanto que os registros com chaves maiores estão na subárvore da direita onde inserções, remoções e buscas possuem número de comparações proporcional à altura da árvore. As buscas são eficientes em árvores balanceadas



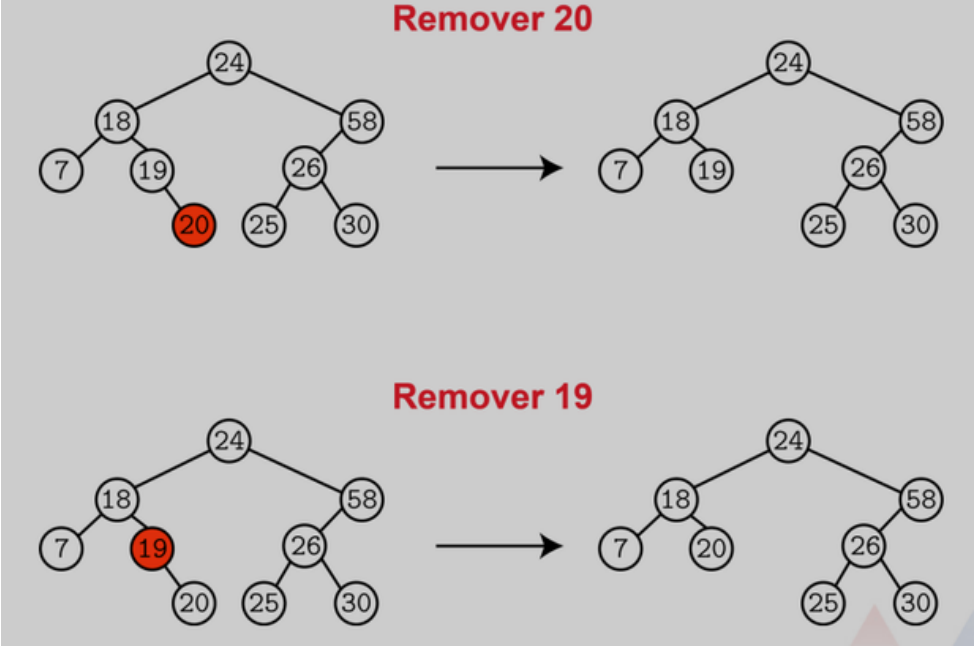
Se chave igual a nó, então achamos. Se chave maior que nó, pesquisamos na subárvore da direita. Caso contrário, na da esquerda. Se alcançarmos um nó nulo, então paramos.

INSERÇÃO ABB



Supondo que não permitimos duplicação na árvore, inserimos apenas se o elemento não existe. Nesse caso, basta inserir o elemento na posição que ele estaria se fosse buscado.

REMOÇÃO ABB



Se o nó a ser removido não possui filhos, simplesmente remova.

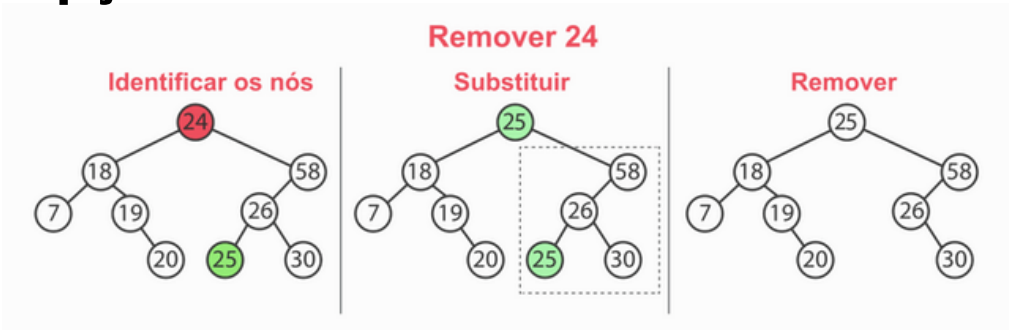
Se o nó possui um filho, então remova e coloque o filho no lugar.

Se o nó possui mais de um filho?

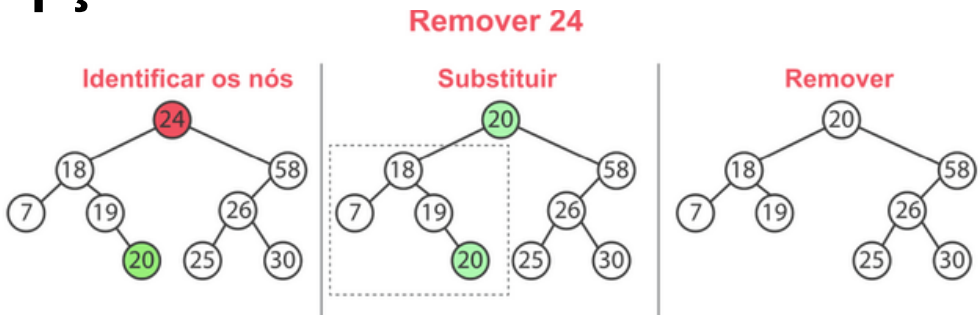
Opção 1:
Se o nó possui mais de um filho, então substitua pelo sucessor lógico antes de remover. O sucessor lógico é sempre o elemento mais à esquerda na subárvore da direita.

Opção 2:
Se o nó possui mais de um filho, então substitua pelo antecessor (ou predecessor) lógico antes de remover. O predecessor lógico é sempre o elemento mais à direita na subárvore da esquerda

Opção 1



Opção 2

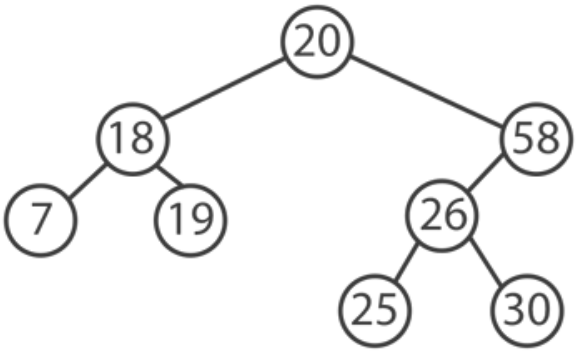


Sempre visitamos a subárvore da esquerda antes de visitarmos a subárvore da direita.

Pré-ordem: visitamos, a partir da raiz, primeiramente o nó raiz, depois os nós da esquerda, depois os da direita.

Pós-ordem: visitamos, a partir da raiz, primeiramente os nó da esquerda, depois os nós da direita e depois concluímos visitando o nó raiz.

In-ordem: visitamos, a partir da raiz, primeiramente os nós da esquerda, depois visitamos o nó raiz, e depois concluímos visitando os nós da direita.



Pré-ordem:	20	18	7	19	58	26	25	30
Pós-ordem:	7	19	18	25	30	26	58	20
In-ordem:	7	18	19	20	25	26	30	58

Árvores binárias de busca são estruturas fundamentais usadas para construir outras estruturas.
Em geral, podem ser usadas em qualquer situação em que queremos organizar os dados por meio de uma chave usada nas buscas.
Quando inserções e remoções são frequentes, **são melhores** que arranjos ordenados.

APLICAÇÃO: Vamos supor que queremos organizar alunos em uma estrutura e, posteriormente, fazer buscas pelo registro acadêmico (ra)

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

class Aluno{
private :
  int        ra;
  std::string nome;
public:
  Aluno();
  Aluno(int ra, std::string nome);
  string getNome() const;
  int getRa() const;
};
```

```
#include "aluno.h"

Aluno::Aluno(){
  this->ra   = -1;
  this->nome = "";
};
Aluno::Aluno(int ra, std::string nome){
  this->ra    = ra;
  this->nome  = nome;
}
string Aluno::getNome() const {
  return nome;
}
int Aluno::getRa() const{
  return ra;
}
```

- Entender o formalismo de grafos;
- Conhecer as formas de representação computacional de grafos;
- Entender a implementação da estrutura de dados grafos usando a representação interna como uma matriz de adjacências.

UNIVESP Universidade Virtual do Estado de São Paulo

SEMANA 6

ÁRVORE AVL

Uma árvore binária T é denominada AVL quando, para qualquer nó de T, as alturas de suas duas subárvores, esquerda e direita, diferem em módulo de até uma unidade. Nesse caso, v é um nó regulado. Em contrapartida, um nó que não satisfaça essa condição de altura é denominado desregulado, e uma árvore que contenha um nó nessas condições é também chamada desregulada.

Árvores AVLs propõem uma modificação nos algoritmos de inserção e remoção para garantir o balanceamento da árvore.a fim de usar árvores binárias de busca, com a garantia de que as operações serão sempre eficientes.

FATOR DE BALANCEAMENTO

É a diferença de altura entre as subárvores da direita e da esquerda. Em árvores AVL deve ficar entre 1 e -1.
Para calcular:

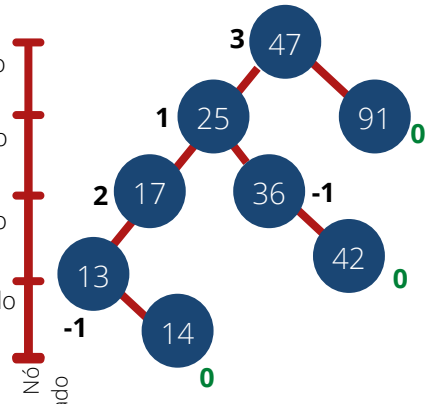
folhas possuem fator de balanceamento 0 para os outros pensar em níveis
Se algum nó violar a propriedade do fator de balanceamento após uma inserção ou remoção, uma rotação deve ser feita.

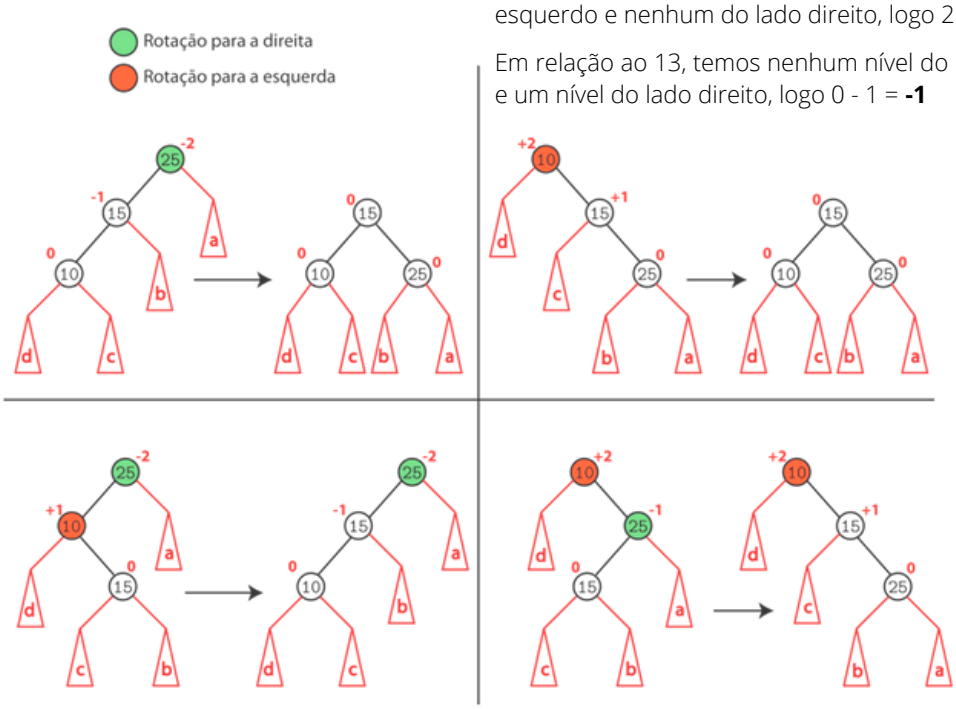
Em relação ao 47 (raíz), temos quatro níveis abaixo do lado esquerdo e um do lado direito, logo 4 - 1 = **3**

Em relação ao 25, temos três níveis abaixo do lado esquerdo e dois do lado direito, logo 3 - 2 = **1**

Em relação ao 17, temos dois níveis abaixo do lado esquerdo e nenhum do lado direito, logo 2 - 0 = **2**

Em relação ao 13, temos nenhum nível do lado esquerdo e um nível do lado direito, logo 0 - 1 = **-1**





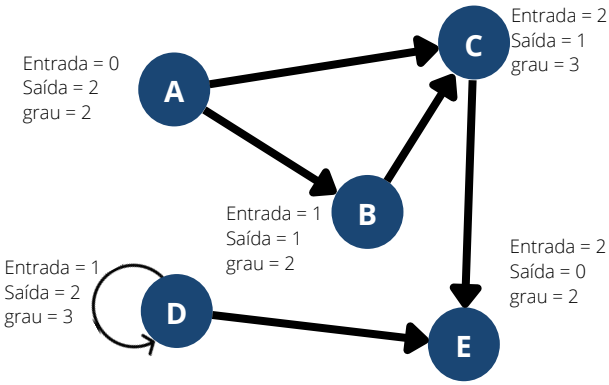
Nó desbalanceado	Filho do Nó desbalanceado	Tipo de Rotação
+2	+1	Simples à esquerda
+2	0	Simples à esquerda
+2	-1	Dupla, com filho para a direita e pai para a esquerda
-2	+1	Dupla, com filho para a esquerda e pai para a direita
-2	0	Simples à direita
-2	-1	Simples à direita

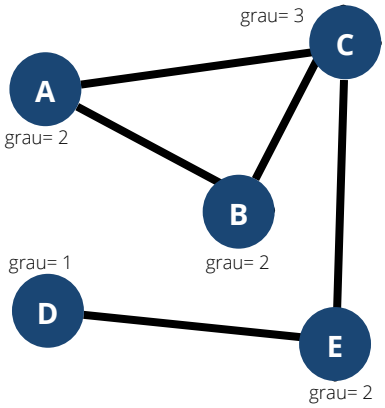
GRAFOS

Representam os casos onde um filho pode ter mais de um pai.
Um grafo G = (V, E) é uma estrutura formada por um conjunto V = (v1 , v2 , ..., vn) de vértices e um conjunto E = (e1 , e2 , ..., em) de arestas, onde <u>cada aresta é um par de vértices.</u>

GRAFOS DIRECIONADOS

Se as arestas são pares ordenados de vértices, saindo de um em direção ao outro: ei = (vj , vk), onde vj , vk ∈ E.
o grau é o número de arestas que saem do vértice (grau de saída) mais o número de arestas que chegam (grau de entrada).





GRAFOS NÃO DIRECIONADOS

Se as relações representadas pelas arestas não têm sentido, ou seja, arestas podem ser seguidas em qualquer direção: ei = {vj , vk }, onde vj , vk ∈ E.
O grau de um vértice é o número de arestas que incidem nele. Note que self-loops não são permitidos.

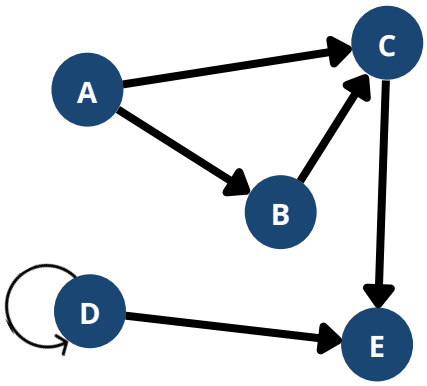
REPRESENTAÇÃO

Matriz de adjacência: Seja G = (V, E) um grafo com n vértices, uma matriz de adjacências A é uma matriz n x n tal que:
A[i, j] = 1, se houver uma aresta indo do vértice i para o vértice j.
A[i,j] = 0, caso contrário.

- Alocaremos espaço para a matriz inteira no momento da declaração da matriz, antes de sabermos o número de vértices e o número de arestas.
- Ocupam o mesmo espaço em grafos esparsos e densos.
- É melhor para testar se existe uma aresta entre dois vértices, dados os índices.
- É melhor para encontrar os predecessores de um nó pois basta olhar a coluna do nó na matriz.

Lista de adjacência: um grafo com n vértices consiste de um arranjo de n listas encadeadas, uma para cada vértice no grafo. • Para cada vértice u, a lista de u contém todos os vizinhos de u.

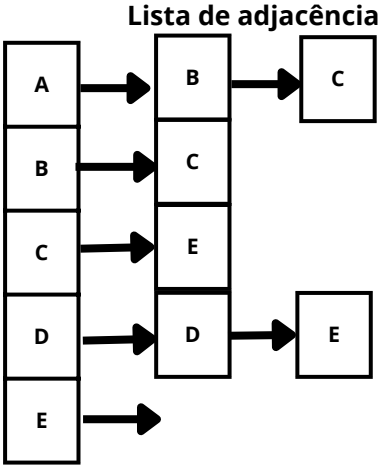
- Buscas são melhores com listas de adjacências, pois já temos os adjacentes de um nó.



Matriz de adjacência

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	1	0	0
C	0	0	0	0	1
D	0	0	0	1	1
E	0	0	0	0	0

Lista de adjacência



- Compreender como ocorre o percorrimento em um grafo por todos os nós, de maneira sistemática, sem entrar em looping infinito.
- Conhecer o algoritmo PageRank, o que inclui os conceitos e a implementação na linguagem C++.

Algoritmos para problemas diversos muitas vezes recaem em algum tipo de visitação sobre os vértices Entender como isso acontece ajuda a evitar loops infinitos. Dois métodos utilizados para isso são a BUSCA EM PROFUNDIDADE E BUSCA EM LARGURA.

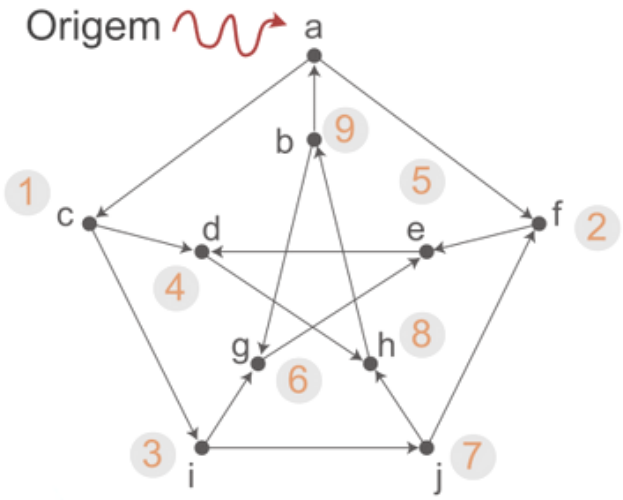
BUSCA EM PROFUNDIDADE: A estratégia consiste em se aprofundar no grafo sempre que possível. Se estamos em um ponto do grafo e ainda há uma caminho não percorrido, seguimos esse caminho. Se estamos em um ponto do grafo e já percorremos tudo ao redor, voltamos para o vértice anterior (backtracking) procurando caminhos não explorados.

O algoritmo é finalizado quando encontramos o que queríamos ou visitamos todos os vértices e não achamos nada. Usamos esse raciocínio em um labirinto. Usa PILHA para implementação.

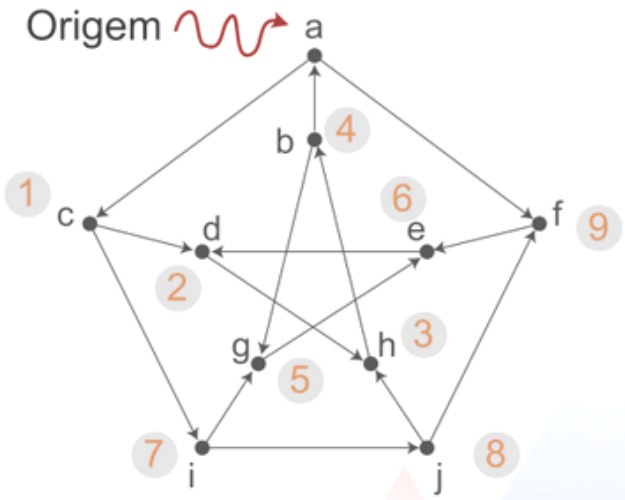
BUSCA EM LARGURA: A estratégia consiste em explorar sem se afastar tanto do ponto inicial. Primeiramente, seguimos um caminho próximo da origem. Se não acharmos o que queríamos, voltamos para o início e tentamos outro caminho . Ao achar o que queríamos, garantimos que sabemos uma maneira rápida de chegar até ele. Em geral, só verificamos vértices a uma distância k+1 se todos os vértices de distância k já tiverem sido visitados.

Intuitivamente, usamos essa ideia quando queremos explorar o ambiente e encontrar um caminho curto até um ponto. Usa FILA para implementação.

Largura



Profundidade



Aplicação Direta de Grafos: PAGERANK

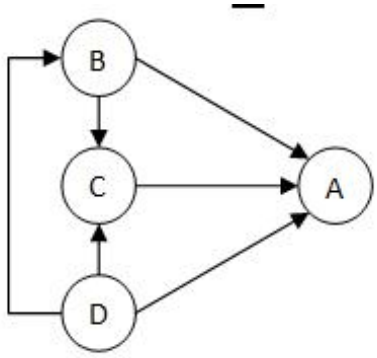
O PageRank é uma métrica criada pelos fundadores do Google e nomeada em referência a Larry Page. A métrica é usada para avaliar a relevância de páginas Web. A métrica estima uma "popularidade" com a quantidade e qualidade dos links para uma página.

- Cada link que uma página p recebe de outras páginas é um voto de suporte, sendo esse voto utilizado para computar o PageRank. Receber links aumenta a autoridade de p.
- Receber links de páginas com PageRank alto é melhor do que links de páginas com PageRank baixo. O voto de suporte de links vindos de páginas com PageRank alto é maior, dado que possuem mais autoridade.

- Páginas que possuem muitos links para outras páginas fornecem um peso menor do que páginas com poucos links. Uma página que recomenda demais deve ser levada menos em conta do que páginas que recomendam menos.

MODELO SIMPLIFICADO

- Links de uma página para si mesma serão Ignorados. Isso faz com que o grafo que representará as páginas web não tenha self-loops.
- Múltiplos links de uma página para outra serão tratados como apenas um link.
- Os PageRanks transferidos de uma página para outra, em cada iteração, são igualmente distribuídos entre todos os links de saída.
- Não existem páginas sem links, pois isso impediria que o PageRank da página fosse distribuído.
- Os PageRanks são uma distribuição de probabilidade. A soma dos valores resulta em 1 (um).
- Uma interpretação dos valores seria que queremos entender em qual página alguém que navega na internet chegaria, clicando em links ao acaso.



No caso da rede em cima, na segunda iteração, o valor de B é transferido metade para A (0,125) e outra metade para C (0,125). Como D referencia 3 páginas, o seu valor a transferir é dividido por três, neste caso o PageRank de A recebe os seguintes valores.

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1} + \frac{PR(D)}{3}.$$

FATOR DE AMORTECIMENTO

- Uma nova variável chamada de fator de amortecimento (damping factor) foi inserida no modelo para superar os problemas de ciclo (rank sink) e páginas sem ligações.
- O fator de amortecimento representa a ideia de que um usuário que navega ao acaso eventualmente parará de clicar em links para ir para outro lugar.
- O fator de amortecimento seria a probabilidade de continuar seguindo os links, recebendo um valor entre 0 (zero) e 1 (um).
- Em geral, o fator de amortecimento é configurado como 0.85. Nesse caso, todas as páginas recebem a mesma chance de serem visitadas ao acaso.

$$PR(p,t) = \frac{1-d}{N} + d\left(\sum_{p' \in \zeta(p)} \frac{PR(p', t-1)}{NumLinks(p')}\right)$$

UNIVESP
Universidade Virtual do Estado de São Paulo

ATIVIDADES AVALIATIVAS

void swap(int a, int b)	Nesta função há a passagem dos dados contidos em a e b por valor, denotada pela notação do argumento da função (int a, int b).
void swap(int &a, int &b)	Aqui há a passagem por referência;
int swap(a,b,c)	Aqui há a passagem por ponteiros.

```
char *ourCode = new char;
*ourCode = 'a';
char *myCode = new char;
*myCode = 'b';
myCode = ourCode;
```

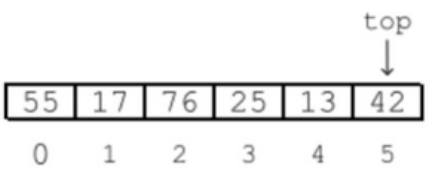
Trecho de código que apresenta um vazamento de memória, situação em que perdemos acesso a uma região de memória, mas não a desalocamos. Note que, na última linha de código, myCode passou a apontar para a mesma região de memória de ourCode. Entretanto, a região de memória para onde myCode apontava anteriormente não foi desalocada, o que caracteriza um vazamento.

delete [] c No caso, para desalocar todos os elementos do vetor em C++ utiliza-se [] (colchetes vazios).

delete **c Aqui apenas o primeiro elemento de memória do vetor (ponteiro-base) será desalocado;

delete c Aqui implicará um erro pois delete espera um ponteiro e não o valor por ele referenciado

Em C++ a alocação e desalocação dinâmica de memória, i.e., a determinação de endereços de memória em tempo de execução, é dada pelos operadores new e delete, respectivamente.
Para especificar uma nova classe, uma variável/função pelo tipo/tipo de retorno (int, float, bool...) e em para alocar a memória estaticamente, especifica-se uma nova classe pelo operador class.



Pop()
55 17 76 25 13

Pop()
55 17 76 25

Push(45)
55 17 76 25 45

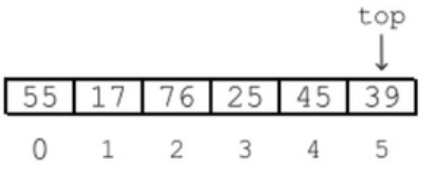
Push(11)
55 17 76 25 45 11

Pop()
55 17 76 25 45

Push(90)
55 17 76 25 45 90

Pop()
55 17 76 25 45

Push(39)
55 17 76 25 45 39



Uma pilha é uma estrutura de dados linear cujas operações de inserção e remoção ocorrem em apenas uma posição, chamada de início (ou topo) da pilha.

```
Stack::Stack()
{
  length = 0;
  structure = new ItemType[MAX_ITEMS];
}

void Stack::push(ItemType item) {
    if(!isFull()){
        structure[length] = item;
        length++;
    } else {
        throw "Stack is already full!";
    }
}
```

Método push no tipo abstrato de dados pilha da classe Stack é necessário saber se a pilha não está cheia antes de nela se inserir um elemento, assim como incrementar o índice que aponta para o topo da pilha, no caso, a variável length.

Módulo (%)

Para evitar desperdício de memória e a determinação incorreta de uma fila cheia e torná-la circular, a operação m´oduloé feita sobre os índices (back e front) ao se inserir ou remover os elementos de uma fila. Imagina-se o vetor da fila como uma estrutura circular, no qual o resto da divisão (módulo) dos índices back e front pelo número máximo de elementos (tamanho) N da fila retorna a mesma posição de memória a cada N inserções ou remoções

```
void Time::setMinute(int minute) {
  this->minute = minute;
}
```

::

O operador apresentado no código é o operador de resolução de escopo

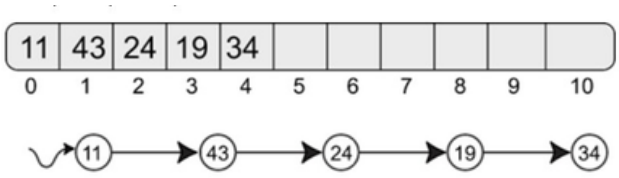
A unidade básica da Programação Orientada a Objetos (POO) é a classe, que encapsula atributos (variáveis) e métodos (funções) em uma camada de abstração. Um objeto é uma instância de uma classe (um tipo) de objetos.

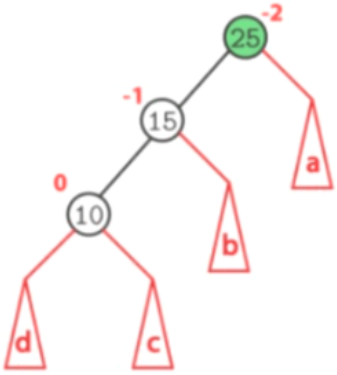
```
if (rear = NULL) {
    front = new_node;
}
```

Verificar se a fila está vazia, se sim, o primeiro nó é o nó recém-criado. Se o ponteiro que aponta para o nó do fim da fila (rear) apontar para o vazio, i.e., para NULL, significa que a fila está vazia e o primeiro nó é o recém-criado. Caso contrário, i.e., a fila não está vazia, precisamos inserir o nó recém-criado no fim da fila, ou seja, o ponteiro do nó do fim da fila aponta para o nó recém-criado

Pilhas são estruturas lineares que podem ser implementadas tanto com vetores quanto com listas encadeadas. A seguir, ilustramos uma estrutura de pilha implementada com vetor e outra implementada com lista encadeada.

No caso das duas estruturas, notamos que possuem os mesmos elementos. Entretanto, uma das pilhas está na ordem inversa da outra. Isso ocorre porque, no caso da pilha implementada como vetor, a cabeça da pilha é o último elemento preenchido no vetor, em outras palavras, a cabeça da pilha está no elemento 34 na implementação em vetor. Por outro lado, a cabeça da pilha na lista encadeada está apontando para o elemento 11. Como a operação Pop() devolve o elemento da cabeça da pilha, é devolvido o 34 da pilha em vetor e o 11 da pilha em lista encadeada





A operação para manter a árvore balanceada, dev ser feita uma rotação para a direita.

A definição de fator de balanceamento refere-se a diferença de altura entre as subárvores da direita e da esquerda.

Em uma árvore AVL, se imediatamente após uma inserção um nó terminar com fator de balanceamento +2, e o filho da direita possui fator de balanceamento +1, então é feita uma rotação simples à esquerda no nó pai para restaurar as propriedades da árvore AVL.

Lista Sequencial: Acesso (leitura) em tempo constante e p ossibilidade de busca de elementos em algoritmo de busca binária (complexidade temporal O(log N)).

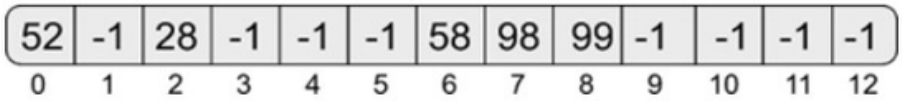
Lista Encadeada: Alocação de memória em tempo de execução e Inserção e remoção em tempo constante.

Na representação usando vetor, os elementos da estrutura linear estarão em posições contíguas de memória. Nesse caso, a ordem lógica dos elementos é semelhante à ordem física em memória. Na representação usando listas encadeadas, os elementos da estrutura linear estarão dispersos na memória principal e, para manter a ordem lógica, cada um dos elementos terá um ponteiro para a região de memória onde podemos encontrar o seu sucessor. Na estrutura vetor, podemos acessar cada elemento, dado o seu índice, em tempo constante e podemos fazer uso da estrutura para ganhar eficiência. Por exemplo, a organização de uma sequência ordenada como um vetor permite a utilização de busca binária, que executa de maneira eficiente. Já na estrutura em listas encadeadas, não precisamos definir o número de elementos da estrutura no momento da alocação da memória, sendo possível ocupar memória sempre que um novo elemento é inserido, e desalocar memória sempre que um elemento é removido e a manutenção da ordem lógica não exige deslocamento de elementos, bastando a atualização de ponteiros, ou seja, a inserção e a remoção de elementos é feita em tempo constante. Isso difere dos vetores em que podemos necessitar fazer deslocamento de elementos para manter a ordem lógica

A desalocação de memória pelo destrutor afeta apenas o primeiro nó da lista encadeada, o topo da pilha. Caso a pilha seja destruída e tenha mais de um nó, o espaço em memória alocado para esses nós é perdido, i.e., há vazamento de memória. É necessário que o destrutor dessa classe de pilha em lista encadeada percorra a lista e desaloque memória nó por nó:

```
stack::~stack() {

    node_type *pointer;      // Ponteiro temporário


    while (structure != NULL) {

        // Guardar o endereço do nó atual

        pointer = structure;

        // Apontar para o próximo nó da lista (obs: '->' é

        // utilizado para ponteiros ao invés de '.')

        structure = structure->next;

        // Desalocar a memória reservada ao nó

        delete pointer;

    }
```

Uma tabela hash recebe como chave valores inteiros. Internamente, a tabela hash foi implementada como um vetor de tamanho 13, com elementos indexados de 0 a 12.
Para tratamento de colisões, é usado o teste linear. Vamos assumir que a seguir temos uma tabela hash obtida após algumas operações de inserção. Note que "-1" indica uma posição vazia.
Se quisermos inserir o elemento 20, ele será mapeado pela função de espalhamento para a posição onde está o elemento 98. Nesse caso, o teste linear tentaria colocar na posição seguinte, que também está ocupada pelo elemento 99, restando então colocar o elemento 20 na posição 9, que está livre



Se tentarmos inserir o elemento 24, criaremos uma colisão com o elemento 52, sendo que esta colisão será tratada pelo teste linear adicionando o 24 na posição 1. Incorreto, o 24 será mapeado para a posição 11, não gerando nenhuma colisão.

Se removermos o 98 e depois inserirmos o elemento 15, este último ficará na posição 7. Incorreto, o elemento 15 será mapeado para a posição 2, que possui o elemento 28. Nesse caso, o teste linear irá posicionar o elemento 15 na posição 3.

Se inserirmos o elemento 60, ele será fisicamente colocado exatamente na posição indicada pela função de espalhamento. Incorreto, o elemento 60 será mapeado para a posição 8, que já possui o elemento 99. Nesse caso, o elemento 60 será colocado na posição 9.

Observando a configuração atual da tabela hash, podemos concluir que apenas uma colisão ocorreu, dado que apenas um elemento está posicionado em lugar diferente daquele indicado pela função de espalhamento. Incorreto, todos os elementos estão na posição indicada pela função de hash.

Se quisermos inserir o elemento 20, ele será mapeado pela função de espalhamento para a posição onde está o elemento 98. Nesse caso, o teste linear tentaria colocar na posição seguinte, que também está ocupada pelo elemento 99, restando então colocar o elemento 20 na posição 9, que está livre.

Uma tabela recebe chaves do tipo string e armazena os dados internamente como um vetor. A função de espalhamento da tabela Hash utiliza o seguinte procedimento para mapear as strings em inteiros:

1 – Mapeamento de caracteres: os três primeiros caracteres são mapeados em inteiros da forma:

De a até f: mapeado para 1
De g até m: mapeado para 3
De n até s: mapeado para 5
De t até z: mapeado para 7

2 – Os inteiros associados a cada um dos três primeiros caracteres são multiplicados.
3 – O resto da divisão por 11 é computado, dado que o vetor possui tamanho 11.

Dadas as seguintes strings: ULISSES, DANIELLE, LARISSA, e aplicando a função de espalhamento apresentada, indique a alternativa correta que apresenta a string e a posição obtida

$ulisses \Rightarrow u:7, l:3, i:3 \Rightarrow 7 * 3 * 3 = 63 \Rightarrow 63\ mod\ 11 = 8$

$danielle \Rightarrow d:1, a:1, n:5 \Rightarrow 1 * 1 * 5 = 5 \Rightarrow 5\ mod\ 11 = 5$

$larissa \Rightarrow l:3, a:1, r:5 \Rightarrow 3 * 1 * 5 = 15 \Rightarrow 15\ mod\ 11 = 4$

Considere que nesta implementação estamos simplesmente garantindo que não colocaremos um registro fora dos limites do vetor (considera a não existência de colisões). Indique qual é alternativa correta que exemplifica porque Pedro e Paulo não aparecem no vetor depois de inserir todos os alunos, como mostra a Figura 1.

```
int main(){
  Hash alunosHash(10);
   int   ras[7]   = {
     12704, 31300, 1234,
     49001, 52202, 65606,
     91234};
  string nomes[7] = {
    "Pedro", "Raul", "Paulo",
    "Carlos", "Lucas", "Maria",
    "Samanta"};
```

```
0:31300, Raul
1:49001, Carlos
2:52202, Lucas
3:-1,
4:91234, Samanta
5:-1,
6:65606, Maria
7:-1,
8:-1,
9:-1,
```

Como essa implementação não trata de colisões, Pedro foi inserido primeiro na posição 4, depois o Paulo foi inserido na posição 4 e finalmente Samantha foi inserida na posição 4. Pedro e Paulo forma apagados para dar lugar a outros alunos por causa das colisões

Considere que nesta implementação estamos simplesmente garantindo que não colocaremos um registro fora dos limites do vetor (considera a não existência de coalisões)

```
14   int Hash::getHash(Aluno aluno){
15
16   }
```

```
int Hash::getHash(Aluno aluno){
  return aluno.getRa() % max_items;
}
```

Na função getHash é considerado o RA do aluno. A função calcula a partir do RA% max_items gerando um valor entre 0 e número máximo de itens o vetor. A função assume a não existência de coalisões

Seja uma tabela hash implementada como um vetor de tamanho 13, com elementos indexados de 0 a 12. Nesse caso, para obter a posição a partir de uma chave, a função de espalhamento computa o resto da divisão da chave por 13. Indicar a alternativa correta que apresenta a chave (92,3,24) e o índice resultado da função de espalhamento

$$92 \rightarrow h(92) = 92 \; mod \; 13 = 1$$

$$3 \rightarrow h(3) = 3 \; mod \; 13 = 3$$

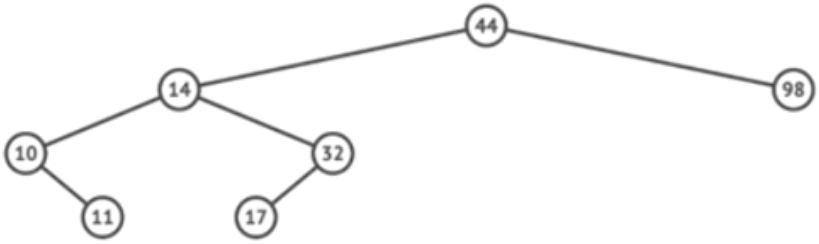
$$24 \rightarrow h(24) = 24 \; mod \; 13 = 11$$

A função hash(chave) deve ser determinística. Para uma determinada chave a função sempre retorna o mesmo valor de hash. Por ser utilizada como uma função de indexação, a função de hash deve sempre retornar um valor de hash dentro dos limites da tabela [0,N−1], onde N é o tamanho da tabela. O método aproveita a possibilidade de acesso randômico à memória para alcançar uma complexidade média por operação de O(1), sendo o pior caso, entretanto, O(n).

FILA: estrutura linear em que o primeiro elemento a entrar tem que ser o primeiro a sair.

Tabela HASH: estrutura que mapeia a chave de busca diretamente para um endereço de memória (endereço base).

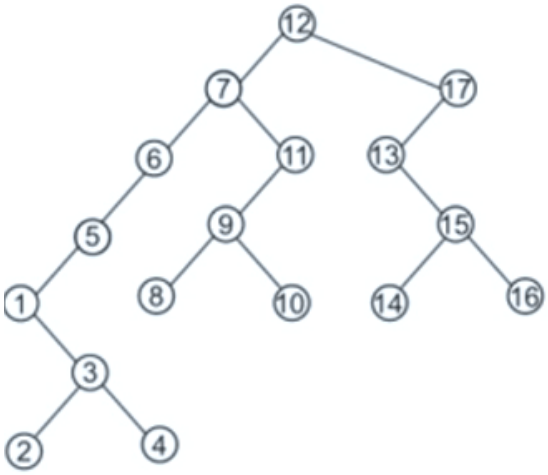
PILHA: estrutura linear em que as inserções e remoções ocorrem na mesma posição



Pré-ordem: 44, 14, 10, 11, 32, 17, 98

Pós-ordem: 11, 10, 17, 32, 14, 98, 44

Em uma árvore binária de busca inicialmente vazia, inserimos os elementos 69, 32, 73, 16 e 30 nessa ordem. Qual das alternativas a seguir esquematiza a estrutura da árvore resultante, onde r é o nó raiz?

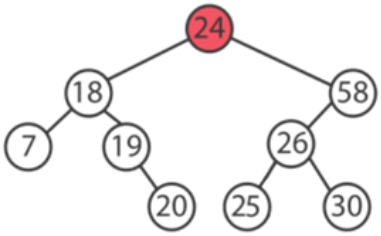


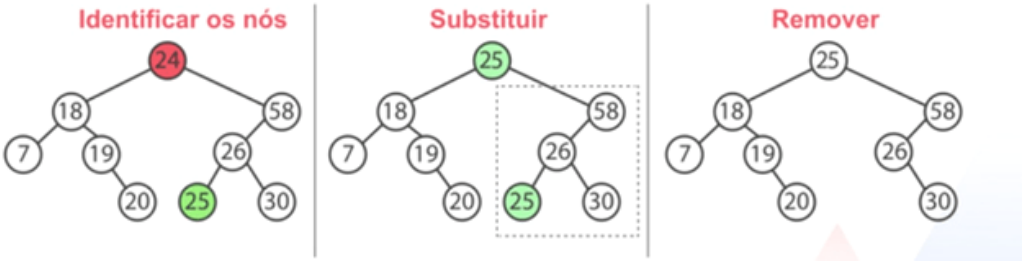
Pré-ordem: 12 7 6 5 1 3 2 4 11 9 8 10 17 13 15 14 16

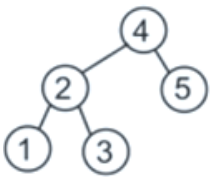
Pós-ordem: 2 4 3 1 5 6 8 10 9 11 7 14 16 15 13 17 12

Dada a seguinte árvore e identificando o nó 24 como aquele que será removido, indicar qual é a data (o valor- chamado de sucessor lógico) que o método getSuccessor vai proporcionar para fazer a substituição do nó 24 nessa árvore

```
void SearchTree::getSuccessor(NodeType* tree, Aluno& data)
{
  tree = tree->direita;
  while (tree->esquerda != NULL)
    tree = tree->esquerda;
  data = tree->aluno;
}
```







Supondo que temos uma estrutura de árvore binária de busca, como a mostrada a seguir, e queremos substituir A,B,C,D e E com números entre 1 e 5 (1,2,3,4,5)

A árvore binária de busca são estruturas fundamentais usadas para construir outras estruturas. Elas podem ser usadas em qualquer situação em que queremos organizar os dados por meio de uma chave usada nas buscas.