Árvores binárias de busca

Tempo para Inserção, Remoção e Busca

Usando Listas Duplamente Ligadas:
- Podemos inserir e remover em $O(1)$
- Mas buscar demora $O(n)$

Se usarmos vetores não-ordenados:
- Podemos inserir e remover em $O(1)$
  - insira no final
  - para remover, troque com o último e remova o último
- Mas buscar demora $O(n)$

Se usarmos vetores ordenados:
- Podemos buscar em $O(\lg n)$
- Mas inserir e remover leva $O(n)$

Veremos árvores binárias de busca
- primeiro uma versão simples, depois uma sofisticada
- versão sofisticada: três operações levam $O(\lg n)$

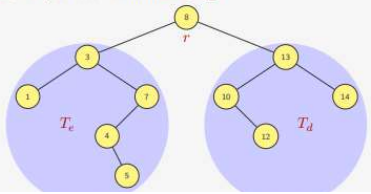
Listas duplamente ligadas: Ou inserimos no início ou no final (por isso custaria O(1)). Mas para fazer a busca, teríamos que percorrer a lista inteira pois poderiam estar em qualquer posição.

Vetores ordenados: Como já sabemos mais ou menos onde os dados estarão, custará O(logn), mas termos que passar pelo vetor todo para saber onde iremos inserir ou remover para não perder a propriedade de ordenação, por isso custa O(n)

Árvore Binária de Busca

Uma Árvore Binária de Busca (ABB) é uma árvore binária em que cada nó contém um elemento de um conjunto ordenável

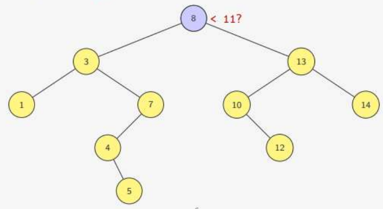
Cada nó r, com subárvores esquerda T_e e direita T_d satisfaz a seguinte propriedade:
1. $e < r$ para todo elemento $e \in T_e$
2. $d > r$ para todo elemento $d \in T_d$



Busca por um valor

A ideia é semelhante a da busca binária:
- Ou o valor a ser buscado está na raiz da árvore
- Ou é menor do que o valor da raiz
  - Se estiver na árvore, está na subárvore esquerda
- Ou é maior do que o valor da raiz
  - Se estiver na árvore, está na subárvore direita

Ex: Buscando por 11



Lógica das perguntas:

1- Verificar se a raiz é igual o valor que estamos buscando

2- Perguntar se o valor da raiz é menor que o valor que estamos buscando, caso for maior vamos para a subárvore da direita, se for menor vamos para a esquerda.

3- Nesse caso vamos fazer a mesma pergunta agora com o nó seguinte (no caso o 13), como ele não é menor que o valor buscado vamos para a subárvore da esquerda agora.

4- Agora vamos fazer a mesma etapa com o 10, como ele é menor que o 11 vamos para a direita...agora chegamos no 12, como o 12 é maior que o 11 ele teoricamente estaria na esquerda do 12 mas ele não está na árvore (nesse caso ele retornaria null → caso fossemos inserir o 11 ele estaria nesse local do null)

Busca

Versão recursiva:

```
p_no buscar(p_no raiz, int chave) {
  if (raiz == NULL || chave == raiz->chave)
    return raiz;
  if (chave < raiz->chave)
    return buscar(raiz->esq, chave);
  else
    return buscar(raiz->dir, chave);
}
```

Versão iterativa:

```
p_no buscar_iterativo(p_no raiz, int chave) {
  while (raiz != NULL && chave != raiz->chave)
    if (chave < raiz->chave)
      raiz = raiz->esq;
    else
      raiz = raiz->dir;
  return raiz;
}
```

*

TAD - Árvores de Busca Binária

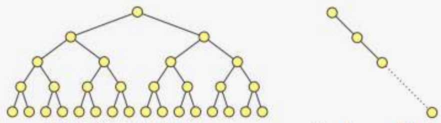
```
typedef struct No {
    int chave;
    struct No *esq, *dir, *pai; /*pai é opcional, usado em
    sucessor e antecessor*/
} No;

typedef No * p_no;

p_no criar_arvore();

void destruir_arvore(p_no raiz);

p_no inserir(p_no raiz, int chave);

p_no remover(p_no raiz, int chave);

p_no buscar(p_no raiz, int chave);

p_no minimo(p_no raiz);

p_no maximo(p_no raiz);

p_no sucessor(p_no x);

p_no antecessor(p_no x);
```

4

Eficiência da busca

Qual é o tempo da busca?
- depende da forma da árvore...

Ex: 31 nós



Melhor árvore: $O(\lg n)$ Pior árvore: $O(n)$

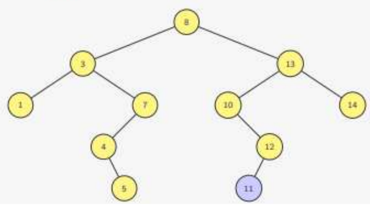
Caso médio: em uma árvore com n elementos adicionados aleatoriamente, a busca demora (em média) $O(\lg n)$

A pior árvore vai ter todos os elementos à esquerda ou todos à direita → No caso do pior cenário possível, seria uma árvore ordenada de forma crescente, pois com a lógica de perguntas teríamos que ir à direita até o final da árvore. Em caso de qualquer outro tipo de árvore nós não teríamos que explorar o lado direito e esquerdo pois a lógica das perguntas já nos colocaria para explorar os nós que nos interessam.

Inserindo um valor

Precisamos determinar onde inserir o valor:
- fazemos uma busca pelo valor
- e inserimos onde ele deveria estar

Ex: Inserindo 11



Inserção - implementação

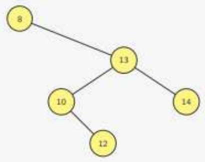
O algoritmo insere na árvore recursivamente

- devolve um ponteiro para a raiz da "nova" árvore
- assim como fizemos com listas ligadas

```c
p_no inserir(p_no raiz, int chave) {
  p_no novo;
  if (raiz == NULL) {
    novo = malloc(sizeof(No));
    novo->esq = novo->dir = NULL;
    novo->chave = chave;
    return novo;
  }
  if (chave < raiz->chave)
    raiz->esq = inserir(raiz->esq, chave);
  else
    raiz->dir = inserir(raiz->dir, chave);
  return raiz;
}
```

Mínimo da Árvore

Onde está o nó com a menor chave de uma árvore?



Quem é o mínimo para essa árvore?

- É a própria raiz

Elementos maiores ficam na subárvore da direita e menores na esquerda. Não há subárvore na esquerda da raiz → o mínimo sempre à esquerda enquanto o máximo mais a direita.

Remoção - Implementação

```c
void remover_sucessor(p_no raiz) {
  p_no min = raiz->dir; /*será o mínimo da subárvore direita*/
  p_no pai = raiz;     /*será o pai de min*/
  while (min->esq != NULL) {
    pai = min;
    min = min->esq;
  }
  if (pai->esq == min)
    pai->esq = min->dir;
  else
    pai->dir = min->dir;
  raiz->chave = min->chave;
}
```

Remoção - Implementação

Versão sem ponteiro para pai e que não libera o nó

```c
p_no remover_rec(p_no raiz, int chave) {
  if (raiz == NULL)
    return NULL;
  if (chave < raiz->chave)
    raiz->esq = remover_rec(raiz->esq, chave);
  else if (chave > raiz->chave)
    raiz->dir = remover_rec(raiz->dir, chave);
  else if (raiz->esq == NULL)
    return raiz->dir;
  else if (raiz->dir == NULL)
    return raiz->esq;
  else
    remover_sucessor(raiz);
  return raiz;
}
```

Mínimo - Implementações

Versão recursiva:

```c
p_no minimo(p_no raiz) {
  if (raiz == NULL || raiz->esq == NULL)
    return raiz;
  return minimo(raiz->esq);
}
```

Versão iterativa:

```c
p_no minimo_iterativo(p_no raiz) {
  while (raiz != NULL && raiz->esq != NULL)
    raiz = raiz->esq;
  return raiz;
}
```

Para encontrar o máximo, basta fazer a operação simétrica

- Se a subárvore direita existir, ela contém o máximo
- Senão, é a própria raiz

Filas de prioridade e heap

Fila de Prioridade

Uma fila de prioridades é uma estrutura de dados com duas operações básicas:

- Inserir um novo elemento
- Remover o elemento com maior *chave* (prioridade)

Uma pilha é como uma fila de prioridades:

- o elemento com maior chave é sempre o último inserido

Uma fila é como uma fila de prioridades:

- o elemento com maior chave é sempre o primeiro inserido

- Na estrutura de uma pilha o último elemento é o de maior prioridade pq ele é o primeiro que sai (a recursão utiliza essa implementação em sua lógica) → LIFO

- Na estrutura de uma fila temos o contrário o primeiro elemento que entra é o primeiro que sai → FIFO

A função troca

Várias vezes iremos trocar dois elementos de posição

Para tanto, vamos usar a seguinte função:

```c
void troca(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}
```

Ou seja, `troca(&v[i], &v[j])` troca os valores de `v[i]` e `v[j]`

Outra opção é colocar diretamente no código da função

- não precisa chamar outra função
- um pouco mais rápido
- código um pouco mais longo e difícil de entender

- A fila de prioridade nada mais é que uma fila comum que permite que elementos sejam adicionados associados com uma prioridade. Cada elemento na fila deve possuir um dado adicional que representa sua prioridade de atendimento.
- A função troca nada mais é que modificar essas prioridades dos elementos, ocorrendo a troca de posição entre eles.

Fila de Prioridade (usando vetores) - TAD

```c
typedef struct {
  char nome[20];
  int chave;
} Item;

typedef struct {
  Item *v;
  int n, tamanho;
} FP;

typedef FP * p_fp;

p_fp criar_filaprio(int tam);

void insere(p_fp fprio, Item item);

Item extrai_maximo(p_fp fprio);

int vazia(p_fp fprio);

int cheia(p_fp fprio);
```

Operações Básicas

```c
p_fp criar_filaprio(int tam) {
  p_fp fprio = malloc(sizeof(FP));
  fprio->v = malloc(tam * sizeof(Item));
  fprio->n = 0;
  fprio->tamanho = tam;
  return fprio;
}
```

```c
void insere(p_fp fprio, Item item) {
  fprio->v[fprio->n] = item;
  fprio->n++;
}
```

```c
Item extrai_maximo(p_fp fprio) {
  int j, max = 0;
  for (j = 1; j < fprio->n; j++)
    if (fprio->v[max].chave < fprio->v[j].chave)
      max = j;
  troca(&(fprio->v[max]), &(fprio->v[fprio->n-1]));
  fprio->n--;
  return fprio->v[fprio->n];
}
```

Insere em $O(1)$, extrai o máximo em $O(n)$

- Se mantiver o vetor ordenado, os tempos se invertem

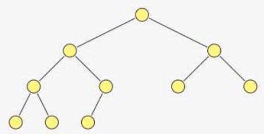
- Se insere em O(1), pois inserimos no início ou no final, e para vermos o máximo, teríamos que passar por todo o vetor. Caso eles estivessem ordenados, a inserção custaria O(n) pois não iríamos inserir nem no ínicio nem no final (teríamos que percorrer o vetor para achar a posição correta dos mesmos para preservar a ordenação) e o máximo iria corresponder o tamanho do vetor já que ele está ordenado.

Árvores Binárias Completas

Uma árvore binária é dita completa se:

- Todos os níveis exceto o último estão cheios
- Os nós do último nível estão o mais à esquerda possível

Exemplo:

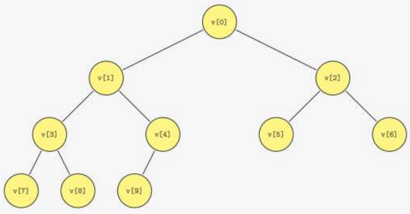


Quantos níveis tem uma árvore binária completa com n nós?

- $\lceil \lg(n+1) \rceil = O(\lg n)$ níveis

Podemos representar tais árvores utilizando também vetores, isto é, não precisaríamos de ponteiros pois utilizaremos a propriedade sequencial dos mesmos.

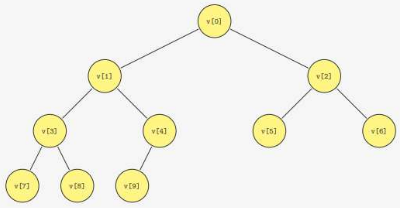
Árvores Binárias Completas e Vetores



Em relação a `v[i]`:

- o filho esquerdo é `v[2*i+1]` e o filho direito é `v[2*i+2]`
- o pai é `v[(i-1)/2]`

Max-Heap

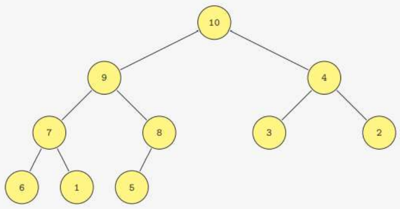


Em um Heap (de máximo):

- Os filhos são menores ou iguais ao pai
- Ou seja, a raiz é o máximo

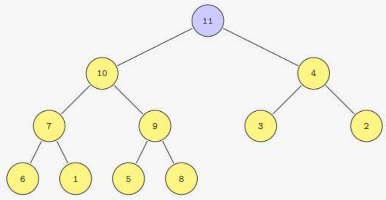
O heap é gerado e mantido no próprio vetor a ser ordenado. Para uma ordenação crescente, deve ser construído um heap máximo (o maior elemento fica na raiz). Para uma ordenação decrescente, deve ser construído um heap mínimo (o menor elemento fica na raiz).

Max-Heap



Note que não é uma árvore binária de busca!

Inserindo no Heap



Basta ir subindo no Heap, trocando com o pai se necessário

- O irmão já é menor que o pai, não precisamos mexer nele

Caso fossemos inserir ou remover teríamos que rearranjar os elementos para mantermos a raiz maior que os filhos e a fins(utilizaremos a função troca em sua implementação)

Inserindo no Heap

```c
void insere(p_fp fprio, Item item) {
  fprio->v[fprio->n] = item;
  fprio->n++;
  sobe_no_heap(fprio, fprio->n - 1);
}

#define PAI(i) ((i-1)/2)

void sobe_no_heap(p_fp fprio, int k) {
  if (k > 0 && fprio->v[PAI(k)].chave < fprio->v[k].chave) {
    troca(&fprio->v[k], &fprio->v[PAI(k)]);
    sobe_no_heap(fprio, PAI(k));
  }
}
```

Tempo de `insere`:

- No máximo subimos até a raiz
- Ou seja, $O(\lg n)$

Extraindo o máximo: Trocamos a raíz com o último elemento do heap. Descemos no Heap arrumando (trocamos o pai com o maior dos dois filhos, se necessário. → em todos os casos ele vai custar O(logn)) por conta da propriedade dos filhos serem menores que os pais (a estrutura garante que tenhamos que fazer uma manipulação apenas entre o pai e algum dos seus filhos pois sabemos que seus irmãos são menores)

Extraindo o Máximo

```c
Item extrai_maximo(p_fp fprio) {
  Item item = fprio->v[0];
  troca(&fprio->v[0], &fprio->v[fprio->n - 1]);
  fprio->n--;
  desce_no_heap(fprio, 0);
  return item;
}

#define F_ESQ(i) (2*i+1) /*Filho esquerdo de i*/
#define F_DIR(i) (2*i+2) /*Filho direito de i*/

void desce_no_heap(p_fp fprio, int k) {
  int maior_filho;
  if (F_ESQ(k) < fprio->n) {
    maior_filho = F_ESQ(k);
    if (F_DIR(k) < fprio->n &&
        fprio->v[F_ESQ(k)].chave < fprio->v[F_DIR(k)].chave)
      maior_filho = F_DIR(k);
    if (fprio->v[k].chave < fprio->v[maior_filho].chave) {
      troca(&fprio->v[k], &fprio->v[maior_filho]);
      desce_no_heap(fprio, maior_filho);
    }
  }
}
```

Tempo de `extrai_maximo`: $O(\lg n)$

Mudando a prioridade de um item

Com o que vimos, é fácil mudar a prioridade de um item

- Se a prioridade aumentar, precisamos subir arrumando
- Se a prioridade diminuir, precisamos descer arrumando

```c
void muda_prioridade(p_fp fprio, int k, int valor) {
  if (fprio->v[k].chave < valor) {
    fprio->v[k].chave = valor;
    sobe_no_heap(fprio, k);
  } else {
    fprio->v[k].chave = valor;
    desce_no_heap(fprio, k);
  }
}
```

Tempo: $O(\lg n)$

- mas precisamos saber a posição do item no heap
- e percorrer o heap para achar o item leva $O(n)$
  - dá para fazer melhor?

Posição do item no heap

Se os itens tiverem um campo `id` com valores de `0` a `n-1`

- Criamos um vetor de `n` posições
- Como parte da `struct` do heap
- Que armazena a posição do item no heap
- Em $O(1)$ encontramos a posição do item no heap

Como modificar os algoritmos para atualizar esse vetor?

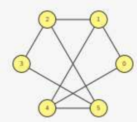
- Toda vez que fizer uma troca, troque também as posições

E se os itens não tiverem esse campo `id`?

- Atribua `ids` aos elementos você mesmo
- Use uma estrutura de dados para encontrar o `id` rapidamente
- Ex: ABBs ou Tabela de Hashing (veremos no futuro)

Grafos

Grafos



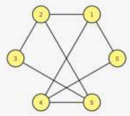
Matematicamente, um grafo G é um par ordenado (V, E)

- V é o conjunto de vértices do grafo
  - Ex: $V = \{0, 1, 2, 3, 4, 5\}$
- E é o conjunto de arestas do grafo
  - Representamos uma aresta ligando $u, v \in V$ como $\{u, v\}$
  - Para toda aresta $\{u, v\}$ em E, temos que $u \neq v$
  - Existe no máximo uma aresta $\{u, v\}$ em E
  - Ex: $E = \Big\{\{0,1\}, \{0,4\}, \{5,3\}, \{1,2\}, \{2,5\}, \{4,5\}, \{3,2\}, \{1,4\}\Big\}$

Matriz de Adjacências

Vamos representar um grafo por uma matriz de adjacências

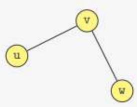
- Se o grafo tem n vértices
- Os vértices serão numerado de 0 a $n-1$
- A matriz de adjacências é $n \times n$
- `adjacencia[u][v] = 1` - u e v são vizinhos
- `adjacencia[u][v] = 0` - u e v não são vizinhos



	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	1
3	0	0	1	0	0	1
4	1	1	0	0	0	1
5	0	0	1	1	1	0

- Matriz simétrica
  - Aij = Aji, ∀ $i, j \epsilon [0, n-1]$
  - Problemas → Se é simétrica, há repetição de dados. O segundo problema é que uma matriz simétrica consome muita memória já que armazena todos os dados (quando aparece muitos 0 → esparsidade da matriz)

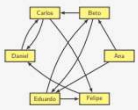
Indicando amigos




```c
void imprime_recomendacoes(p_grafo g, int u) {
  int v, w;
  for (v = 0; v < g->n; v++) {
    if (g->adj[u][v]) {
      for (w = 0; w < g->n; w++) {
        if (g->adj[v][w] && w != u && !g->adj[u][w])
          printf("%d\n", w);
      }
```

Seguindo e sendo seguido

Como representar seguidores em redes sociais?

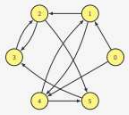


- A Ana segue o Beto e o Eduardo
- Ninguém segue a Ana
- O Daniel é seguido pelo Carlos e pelo Felipe
- O Eduardo segue o Beto que o segue de volta

Grafos dirigidos

Um Grafo dirigido (ou Digrafo)

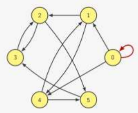
- Tem um conjunto de vértices
- Conectados através de um conjunto de arcos
  - arestas dirigidas, indicando início e fim



Representamos um digrafo visualmente

- com os vértices representados por pontos e
- os arcos representadas por curvas com uma seta na ponta ligando dois vértices

Grafos dirigidos

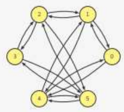


Matematicamente, um digrafo G é um par (V, A)

- V é o conjunto de vértices do grafo
- A é o conjunto de arcos do grafo
  - Representamos um arco ligando $u, v \in V$ como (u, v)
    - u é a cauda ou origem de (u, v)
    - v é a cabeça ou destino de (u, v)
  - Podemos ter laços: arcos da forma (u, u)
  - Existe no máximo um arco (u, v) em A

Grafos e digrafos

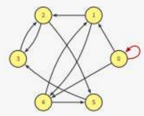
Podemos ver um grafo como um digrafo



Basta considerar cada aresta como dois arcos

- É o que já estamos fazendo na matriz de adjacências
- Ou seja, podemos usar uma matriz de adjacências para representar um digrafo
  - `adjacencia[u][v] == 1`: temos um arco de `u` para `v`
  - pode ser que `adjacencia[u][v] != adjacencia[v][u]`

Grafos dirigidos

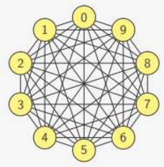


Matematicamente, um digrafo G é um par (V, A)

- V é o conjunto de vértices do grafo
- A é o conjunto de arcos do grafo
  - Representamos um arco ligando $u, v \in V$ como (u, v)
    - u é a cauda ou origem de (u, v)
    - v é a cabeça ou destino de (u, v)
  - Podemos ter laços: arcos da forma (u, u)
  - Existe no máximo um arco (u, v) em A

Número de arestas de um grafo

Quantas arestas pode ter um grafo com n vértices?



Até $\binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$ arestas

Grafos esparsos

Um grafo tem no máximo $n(n-1)/2$ arestas, mas pode ter bem menos...

Facebook tem 2,2 bilhões de usuários ativos/mês

- Uma matriz de adjacências teria $4{,}84 \cdot 10^{18}$ posições
  - 605 petabytes (usando um bit por posição)
- Verificar se duas pessoas são amigas leva $O(1)$
  - supondo que tudo isso coubesse na memória...
- Imprimir todos os amigos de uma pessoa leva $O(n)$
  - Teríamos que percorrer 2,2 bilhões de posições
  - Um usuário comum tem bem menos amigos do que isso...
  - Facebook coloca um limite de 5000 amigos

Grafos esparsos

Dizemos que um grafo é esparso se ele tem "poucas" arestas

- Bem menos do que $n(n-1)/2$

Exemplos:

- Facebook:
  - Cada usuário tem no máximo 5000 amigos
  - O máximo de arestas é $5{,}5 \cdot 10^{12}$
  - Bem menos do que $2{,}4 \cdot 10^{18}$
- Grafos cujos vértices têm o mesmo grau d (constante)
  - O número de arestas é $dn/2 = O(n)$
- Grafos com $O(n \lg n)$ arestas

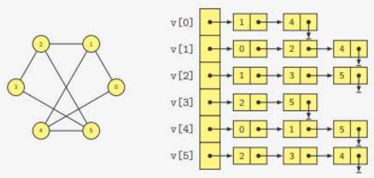
Não dizemos que um grafo com $n(n-1)/20$ arestas é esparso

- O número de arestas não é assintoticamente menor...
- É da mesma ordem de grandeza que n^2...

Listas de Adjacência

Representando um grafo por Listas de Adjacência:

- Temos uma lista ligada para cada vértice
- A lista armazena quais são os vizinhos do vértice



TAD Grafo com Listas de Adjacência

```c
typedef struct No {
  int v;
  struct No *prox;
} No;

typedef No * p_no;

typedef struct {
  p_no *adjacencia;
  int n;
} Grafo;

typedef Grafo * p_grafo;

p_grafo criar_grafo(int n);

void destroi_grafo(p_grafo g);

void insere_aresta(p_grafo g, int u, int v);

void remove_aresta(p_grafo g, int u, int v);

int tem_aresta(p_grafo g, int u, int v);

void imprime_arestas(p_grafo g);
```


Inicialização e Destruição

```c
p_grafo criar_grafo(int n) {
  int i;
  p_grafo g = malloc(sizeof(Grafo));
  g->n = n;
  g->adjacencia = malloc(n * sizeof(p_no));
  for (i = 0; i < n; i++)
    g->adjacencia[i] = NULL;
  return g;
}
```

```c
void libera_lista(p_no lista) {
  if (lista != NULL) {
    libera_lista(lista->prox);
    free(lista);
  }
}
```

```c
void destroi_grafo(p_grafo g) {
  int i;
  for (i = 0; i < g->n; i++)
    libera_lista(g->adjacencia[i]);
  free(g->adjacencia);
  free(g);
}
```


Inserindo uma aresta

```c
p_no insere_na_lista(p_no lista, int v) {
  p_no novo = malloc(sizeof(No));
  novo->v = v;
  novo->prox = lista;
  return novo;
}
```

```c
void insere_aresta(p_grafo g, int u, int v) {
  g->adjacencia[v] = insere_na_lista(g->adjacencia[v], u);
  g->adjacencia[u] = insere_na_lista(g->adjacencia[u], v);
}
```

Removendo uma aresta

```c
p_no remove_da_lista(p_no lista, int v) {
  p_no proximo;
  if (lista == NULL)
    return NULL;
  else if (lista->v == v) {
    proximo = lista->prox;
    free(lista);
    return proximo;
  } else {
    lista->prox = remove_da_lista(lista->prox, v);
    return lista;
  }
}
```

```c
void remove_aresta(p_grafo g, int u, int v) {
  g->adjacencia[u] = remove_da_lista(g->adjacencia[u], v);
  g->adjacencia[v] = remove_da_lista(g->adjacencia[v], u);
}
```

Verificando se tem uma aresta e imprimindo

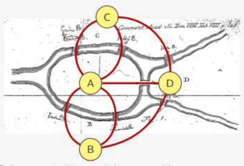
```c
int tem_aresta(p_grafo g, int u, int v) {
  p_no t;
  for (t = g->adjacencia[u]; t != NULL; t = t->prox)
    if (t->v == v)
      return 1;
  return 0;
}
```

```c
void imprime_arestas(p_grafo g) {
  int u;
  p_no t;
  for (u = 0; u < g->n; u++)
    for (t = g->adjacencia[u]; t != NULL; t = t->prox)
      printf("{%d,%d}\n", u, t->v);
}
```

O Problema das Pontes de Königsberg

Königsberg (hoje Kaliningrado, Rússia) tinha 7 pontes

- Acreditava-se que era possível passear por toda a cidade
- Atravessando cada ponte exatamente uma vez



Leonhard Euler, em 1736, modelou o problema como um grafo

- Provou que tal passeio não é possível
- E fundou a Teoria dos Grafos...

É com essa teoria que usamos os percursos de grafos lá que só se pode visitar um vértice por vez, os modelos de "maps de celular" vem disso aí tbm

Comparação Listas e Matrizes

Espaço para o armazenamento:

- Matriz: $O(|V|^2)$
- Listas: $O(|V| + |E|)$

Tempo:

Operação	Matriz	Listas
Inserir	$O(1)$	$O(1)$
Remover	$O(1)$	$O(d(v))$
Aresta existe?	$O(1)$	$O(d(v))$
Percorrer vizinhança	$O(\|V\|)$	$O(d(v))$

As duas permitem representar grafos, digrafos e multigrafos

- mas multigrafos é mais fácil com Listas de Adjacência

Qual usar?

- Depende das operações usadas e se o grafo é esparso

Diferença entre o tempo de operação entre as matrizes e listas, conseguimos perceber que a matriz consegue ser muito mais rápida do as listas.

Operação	Matriz	Listas
Inserir	Matriz vai na posição e marca 1	Insere no início de uma lista
Remover	Acessa a posição e marca com 0	Tem que percorrer a lista de adjacência do vértice v e apagar - d(v) é o grau do vértice
Aresta Existe?	Precisa apenas olhar a posição	Tem que percorrer a lista guiada inteira
Percorrer vizinhança	Precisa olhar para todo o vértice e vê se ele é vizinho ou não	Percorre a lista dele e verifica se tem todos os vizinhos

Qual usar? Se ficar só inserindo e removendo aresta, é melhor a matriz. Se precisar percorrer a vizinhança e o grafo é muito denso, provavelmente vale a pena usar a matriz, entretanto se o grafo é esparso em geral se usa listas de adjacência. Caso precise só inserir e remover aresta em geral, a matriz é melhor.

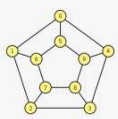
Importância dos Grafos

Grafos são amplamente usados na Computação e na Matemática para a modelagem de problemas:

- Redes Sociais: grafos são a forma de representar uma relação entre duas pessoas
- Mapas: podemos ver o mapa de uma cidade como um grafo e achar o menor caminho entre dois pontos
- Páginas na Internet: links são arcos de uma página para a outra - podemos querer ver qual é a página mais popular
- Redes de Computadores: a topologia de uma rede de computadores é um grafo
- Circuitos Eletrônicos: podemos criar algoritmos para ver se há curto-circuito por exemplo
- etc...

Percurso em Grafos

Caminhos em Grafos



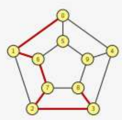
Um caminho de s para t em um grafo é:

- Uma sequência sem repetição de vértices vizinhos
- Começando em s e terminado em t

Por exemplo:

- $0, 1, 6, 7, 2, 3, 8$ é um caminho de 0 para 8
- $0, 5, 8$ não é um caminho de 0 para 8
- $0, 1, 2, 7, 6, 1, 2, 3, 8$ não é um caminho de 0 para 8

Caminhos em Grafos



Formalmente, um caminho de s para t em um grafo é:

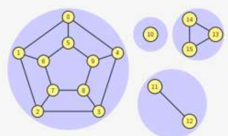
- Uma sequência de vértice $v_0, v_1, \dots, v_k$ onde
- $v_0 = s$ e $v_k = t$
- $\{v_i, v_{i+1}\}$ é uma aresta para todo $0 \leq i \leq k-1$
- $v_i \neq v_j$ para todo $0 \leq i < j \leq k$

k é o comprimento do caminho

- $k = 0$ se e somente se $s = t$

Componentes Conexas

Um grafo pode ter várias "partes"



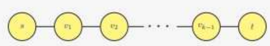
Chamamos essas partes de Componentes Conexas

- Um par de vértices está na mesma componente se e somente se existe caminho entre eles
  - Não há caminho entre vértices de componentes distintas
- Um grafo conexo tem apenas uma componente conexa

Existe caminho entre s e t?

Queremos saber se s e t estão na mesma componente conexa

Se estiverem, existe algum caminho de s até t



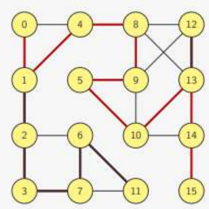
Se existe caminho e $s \neq t$, existe um segundo vértice v_1

- E v_1 é vizinho de s
- Então, ou $v_1 = t$, ou existe um terceiro vértice v_2
  - E v_2 é vizinho de v_1
- E assim por diante...

A dificuldade é acertar qual vizinho v_1 de s devemos usar...

- Solução: testar todos!

Exemplo - Existe caminho de 0 até 15?



Essa é uma busca em profundidade:

- Vá o máximo possível em uma direção
- Se não encontrarmos o vértice, volte o mínimo possível
- E pegue um novo caminho por um vértice não visitado

A ideia é como se fosse labirinto, não podemos visitar um local que já foi visitado → Essa implementação casa bem com a ideia de Pilha ou Recursão para descartar caminhos ou selecionar novas possibilidades. A ideia é visitar o vértice de menor valor que ainda não foi visitado.

Implementação (com Matriz de Adjacências)

```c
int existe_caminho(p_grafo g, int s, int t) {
  int encontrou, i, *visitado = malloc(g->n * sizeof(int));
  for (i = 0; i < g->n; i++)
    visitado[i] = 0;
  encontrou = busca_rec(g, visitado, s, t);
  free(visitado);
  return encontrou;
}
```

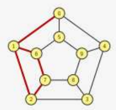
```c
int busca_rec(p_grafo g, int *visitado, int v, int t) {
  int w;
  if (v == t)
    return 1; /*sempre existe caminho de t para t*/
  visitado[v] = 1;
  for (w = 0; w < g->n; w++)
    if (g->adj[v][w] && !visitado[w])
      if (busca_rec(g, visitado, w, t))
        return 1;
  return 0;
}
```

Componentes Conexas (Listas de Adjacência)

```c
int * encontra_componentes(p_grafo g) {
  int s, c = 0, *componentes = malloc(g->n * sizeof(int));
  for (s = 0; s < g->n; s++)
    componentes[s] = -1;
  for (s = 0; s < g->n; s++)
    if (componentes[s] == -1) {
      visita_rec(g, componentes, c, s);
      c++;
    }
  return componentes;
}
```

```c
void visita_rec(p_grafo g, int *componentes, int comp, int v) {
  p_no t;
  componentes[v] = comp;
  for (t = g->adj[v]; t != NULL; t = t->prox)
    if (componentes[t->v] == -1)
      visita_rec(g, componentes, comp, t->v);
}
```

Ciclos em Grafos



Um ciclo em um grafo é:

- Uma sequência de vértices vizinhos sem repetição exceto pelo primeiro e o último vértice que são idênticos

Por exemplo:

- $5, 6, 7, 8, 9, 5$ é um ciclo
- $1, 2, 3$ não é um ciclo
- $1, 2, 7, 6, 1$ é um ciclo
- $1, 2, 7, 6, 1, 0$ não é um ciclo (mas contém um ciclo)

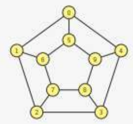
Árvores, Florestas e Subgrafos

Uma árvore é um grafo conexo acíclico

- Uma floresta é um grafo acíclico
- Suas componentes conexas são árvores

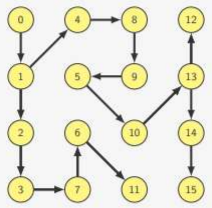
Um subgrafo é um grafo obtido a partir da remoção de vértices e arestas

- Podemos considerar também árvores/florestas que são subgrafos de um grafo dado



Árvores podem ser consideradas conjuntos de árvores. Árvore é um grafo conexo (Só tem uma componente conexa) acíclico (Não tem ciclos nesse gráfico). A diferença dessa árvore para a outra é que ela não é binária e no grafo a árvore não precisa ter raiz

Caminhos de s para outros vértices da componente



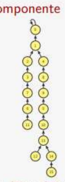
As arestas usadas formam uma árvore!

- Essa árvore dá um caminho de qualquer vértice até a raiz
- Basta ir subindo na árvore

Damos preferência para visitar primeiro sempre o de menor índice.

Caminhos de s para outros vértices da componente

```c
int * encontra_caminhos(p_grafo g, int s) {
  int i, *pai = malloc(g->n * sizeof(int));
  for (i = 0; i < g->n; i++)
    pai[i] = -1;
  busca_em_profundidade(g, pai, s, s);
  return pai;
}
```

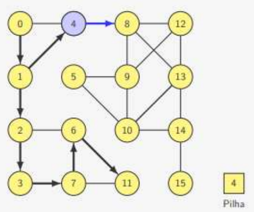


```c
void busca_em_profundidade(p_grafo g, int *pai, int p, int v) {
  p_no t;
  pai[v] = p;
  for(t = g->adj[v]; t != NULL; t = t->prox)
    if (pai[t->v] == -1)
      busca_em_profundidade(g, pai, v, t->v);
}
```

Busca em Profundidade usando uma Pilha

Podemos fazer uma busca em profundidade usando pilha:

- A cada passo, desempilhamos um vértice não visitado
- E inserimos os seus vizinhos não visitados na pilha

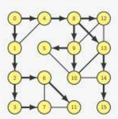


1º Coloca os vizinhos na pilha, ex: os vizinhos do 0 são {1, 4}; 2º Desempilha o vizinho e empilha o próximo vizinho, ex: desempilha o 1 e empilha {2, 4}; 3º No final desempilha tudo e o algoritmo termina.

Implementação

```c
int * busca_em_profundidade(p_grafo g, int s) {
  int w, v;
  int *pai = malloc(g->n * sizeof(int));
  int *visitado = malloc(g->n * sizeof(int));
  p_pilha p = criar_pilha();
  for (v = 0; v < g->n; v++) {
    pai[v] = -1;
    visitado[v] = 0;
  }
  empilhar(p,s);
  pai[s] = s;
  while(!pilha_vazia(p)) {
    v = desempilhar(p);
    visitado[v] = 1;
    for (w = 0; w < g->n; w++)
      if (g->adj[v][w] && !visitado[w]) {
        pai[w] = v;
        empilhar(p, w);
      }
  }
  destroi_pilha(p);
  free(visitado);
  return pai;
}
```

E se tivéssemos usando uma Fila?



Usando uma fila, visitamos primeiro os vértices mais próximos

- Enfileiramos os vizinhos de 0 (que estão a distância 1)
- Desenfileiramos um de seus vizinho
- E enfileiramos os vizinhos deste vértice
  - que estão a distância 2 de 0
- Assim por diante...

A árvore nos dá um caminho mínimo entre raiz e vértice

Implementação da Busca em Largura

```c
int * busca_em_largura(p_grafo g, int s) {
  int w, v;
  int *pai = malloc(g->n * sizeof(int));
  int *visitado = malloc(g->n * sizeof(int));
  p_fila f = criar_fila();
  for (v = 0; v < g->n; v++) {
    pai[v] = -1;
    visitado[v] = 0;
  }
  enfileira(f,s);
  pai[s] = s;
  visitado[s] = 1;
  while(!fila_vazia(f)) {
    v = desenfileira(f);
    for (w = 0; w < g->n; w++)
      if (g->adj[v][w] && !visitado[w]) {
        visitado[w] = 1;/*evita repetição na fila*/
        pai[w] = v;
        enfileira(f, w);
      }
  }
  destroi_fila(f);
  free(visitado);
  return pai;
}
```


Tempo para fazer a busca

Quanto tempo demora para fazer uma busca?

- em profundidade ou em largura
- em um grafo com n vértices e m arestas

Suponha que inserir e remover de pilha/fila leva $O(1)$

- Podemos usar vetores ou listas ligadas

A busca percorre todos os vértices

- E empilha/enfileira seus vizinhos não visitados
- Se usarmos uma Matriz de Adjacências, leva $O(n^2)$

E se usarmos Listas de Adjacência?

- Cada aresta é analisada apenas duas vezes
- Gastamos tempo $O(\max\{n, m\}) = O(n + m)$
  - Linear no tamanho do grafo