Onde estamos ...

Capítulo 04 – Filas

Pontos fundamentais a serem cobertos:

1. Contexto e motivação
2. Definição
3. Implementações
4. Exercícios

Introdução

- Assim como a estrutura de dados Pilha, Fila é outra estrutura de dados bastante utilizada em computação.
- Um exemplo é a implementação de uma fila de impressão.
- Se uma impressora é compartilhada por várias máquinas, normalmente adota-se uma estratégia para determinar a ordem de impressão dos documentos.
- A maneira mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos – o primeiro submetido é o primeiro a ser impresso.

Fila

Aplicações

1. Escalonamento de processos na CPU (processos com a mesma prioridade) os quais são executados em ordem de chegada
2. Simulação de filas no mundo real tais como: filas em banco, compra de tickets, etc
3. Multi-programação (*time-sharing*)
4. Transferência assíncrona de dados (IO de arquivos, pipe, sockets)
5. Fila de espera em um *call-center*
6. Encontrar número de atendentes em supermercados e caixas bancários dado uma demanda de pessoas na sala de espera

Fila

Aplicações Indiretas

1. Estrutura de dados auxiliares em algoritmos
2. Componente de outras estruturas

Fila

Definição

Um conjunto ordenado de itens a partir do qual podem-se eliminar itens numa extremidade (chamada de início da fila) e no qual podem-se inserir itens na outra extremidade (chamada final da fila).

Representação

- Os nós de uma fila são armazenados em endereços contínuos.
- A Figura **??** ilustra uma fila com três elementos.

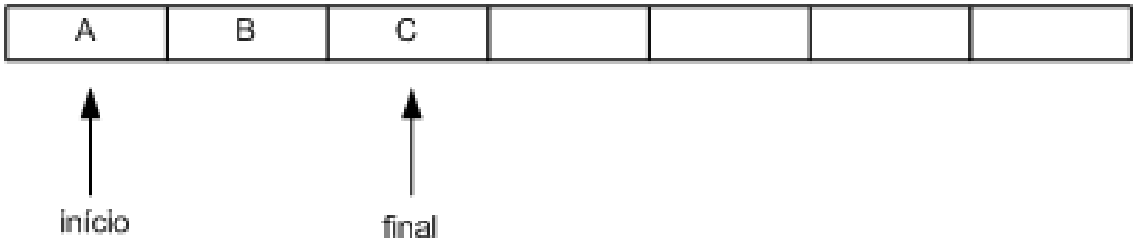


Figura 1: Exemplo de representação de fila.

- Após a retirada de um elemento (*primeiro*) temos:

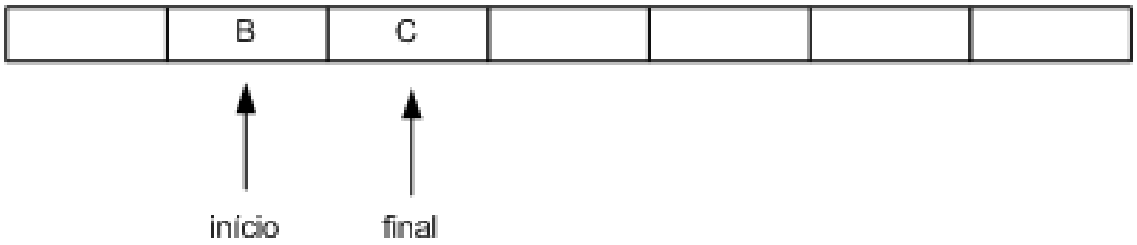


Figura 2: Representação de uma fila após a remoção do elemento "A".

Implementações

Quais as estruturas usadas por filas?

- Baseada num vetor simples – limitada
- Baseada num vetor circular simples – limitada – igualmente poderosa
- Baseada num vetor circular dinâmico – **i**limitada
- Baseada numa lista encadeada (depois de listas)

Representação

- Após a inclusão de dois elementos temos:

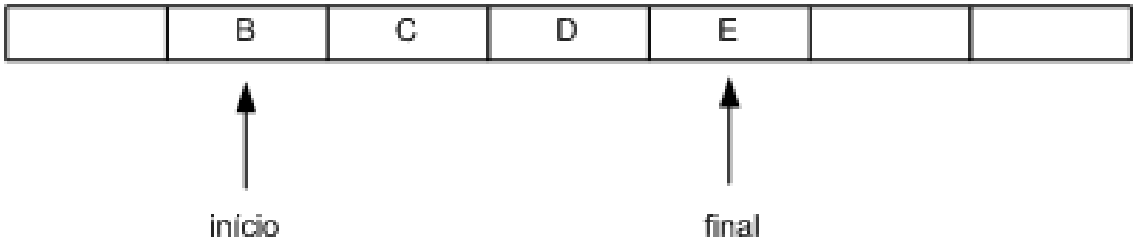


Figura 3: Representação de uma fila após a inclusão de dois elementos "D" e "E".

- Como podemos observar, a operação de inclusão e retirada de um item da fila incorre na mudança do endereço do ponteiro que informa onde é o início e o término da fila.

Representação

- Em uma fila, o primeiro elemento inserido é o primeiro a ser removido.
- Por essa razão, uma fila é chamada **fifo (*first-in first-out*)** – primeiro que entra é o primeiro a sair – ao contrário de uma pilha que é lifo (*last-in, first-out*)
- Para exemplificar a implementação em C, vamos considerar que o conteúdo armazenado na fila é do tipo inteiro.
- Nos exemplos do prof é um outro tipo de dado
- A estrutura de fila possui a seguinte representação:

```
1   struct fila{
2     int elemento[N];
3     int ini;
4     int n; // quantos tem na fila
5   };
6
7 typedef struct fila Fila;
```

Representação

- Trata-se de uma estrutura heterogênea constituída de membros distintos entre si.
- Os membros são as variáveis *ini* e *fim*, que serve para armazenar respectivamente, o início e o fim da fila e o vetor *elemento* de inteiros que armazena os itens da fila.

Operações Primitivas

- As operações básicas que devem ser implementadas em uma estrutura do tipo Fila são:

Operação	**Descrição**
criar()	aloca dinamicamente a estrutura da fila.
insere(f,e)	adiciona um novo elemento *(e)*, no final da fila *f*.
retira(f)	remove o elemento do início da fila *f*.

Tabela 1: Operações básicas da estrutura de dados fila.

Operações Auxiliares

- Além das operações básicas, temos as operações "*auxiliares*". São elas:

Operação	**Descrição**
vazia(f)	informa se a fila está ou não vazia.
libera(f)	destrói a estrutura, e assim libera toda a memória alocada

Tabela 2: Operações auxiliares da estrutura de dados fila.

Interface do Tipo Fila

```c
typedef struct fila Fila;
/* Aloca dinamicamente a estrutura Fila, inicializando seus
 * campos e retorna seu ponteiro. A fila depois de criada
 * estarah vazia.*/
Fila* criar(void);

/* Insere o elemento e no final da fila f, desde que,
 * a fila nao esteja cheia.*/
void insere(Fila* f, int e);

/* Retira o elemento do inicio da fila, e fornece o
 * valor do elemento retirado como retorno, desde que a fila
 * nao esteja vazia*/
int retira(Fila* f);

/*Verifica se a fila f estah vazia*/
int vazia(Fila* f);

/*Libera a memoria alocada pela fila f*/
void libera(Fila* f);
```

Reflexão – Pausa

- Como em pilhas, e se estas estruturas puderem ser reutilizadas em diversos tipos de problemas?
- Ou seja, o reuso das mesmas estruturas de dados em problemas diferentes!
- O que é isto?
- Bem-vindos ao Tipos Abstractos de Dados (TAD ou TDA)
- Coloque as declarações de funções em um arquivo `.h` (Veja stdio.h)
- Coloque as funções em arquivos `.c`
- Deixe os arquivos `.o` para serem *linkeditados* no código relocável!
- Basicamente fizemos isto com o *Makefile*

Implementação de Fila com Vetor

- Assim como nos casos da pilha e lista, a implementação de fila será feita usando um vetor para armazenar os elementos.
- Isso implica, que devemos fixar o número máximo de elementos na fila.
- O processo de inserção e remoção em extremidades opostas fará a fila "*andar*" no vetor.
- Por exemplo, se inserirmos os elementos 8, 7, 4, 3 e depois retiramos dois elementos, a fila não estará mais nas posições iniciais do vetor.

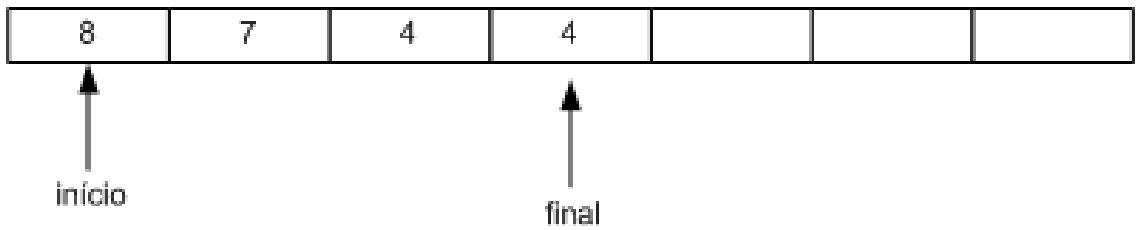


Figura 4: Fila após inserção de quatro elementos (um erro na figura 4/3)

Implementação de Fila com Vetor

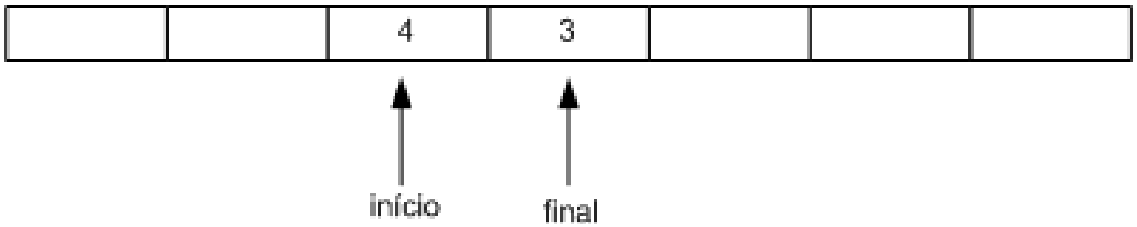


Figura 5: Fila após retirar dois elementos.

- Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada pelo vetor pode chegar a última posição.
- Uma solução seria ao remover um elemento da fila, deslocar a fila inteira no sentido do início do vetor.
- Entretanto, essa método é bastante ineficiente, pois cada retirada implica em deslocar cada elemento restante da fila. Se uma fila tiver 500 ou 1000 elementos, evidentemente esse seria um preço muito alto a pagar.

Implementação de Fila com Vetor

- Para reaproveitar as primeira posições do vetor sem implementar uma "*re-arrumação*" dos elementos, podemos incrementar as posições do vetor de forma "*circular*".
- Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam "*circularmente*".
- Dessa forma, se temos 100 posições no vetor, os índices assumem os seguintes valores:

$$0, 1, 2, 3, \cdots, 98, 99, 0, 1, 2, 3, \cdots, 98, 99, \cdots$$

Porquê um *vetor circular*?

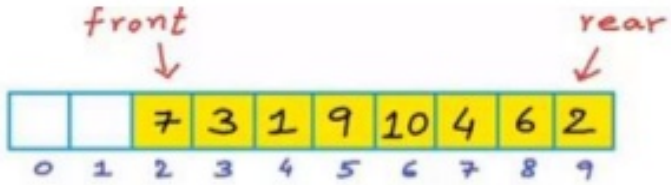
Reflexões

- Tamanho fixo – muito interessante para maioria dos problemas
- Velocidade
- Fácil de manipular
- *Circular* é a manipulação!
- Calcule $f(i+1)$ dado que a cabeça ou fim de estivessem na posição i em um vetor de N posições (adote a convenção no sentido-horário)
- Cálculo com papel e caneta mesmo!

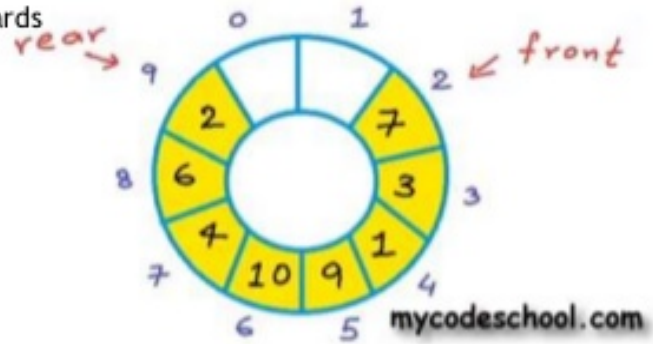
Vetor Circular: Explicações

Queue: implementation Cyclic Array

no end on array
wrapping around
ring buffer / circular buffer
push/pop - O(1)



when item inserted to rair, tails's pointer moves upwards
when item deleted, head's pointer moves downwards



current_position = i
next_position = (i + 1) % N
prev_position = (N + i - 1) % N

19

Vetor Circular: Explicações

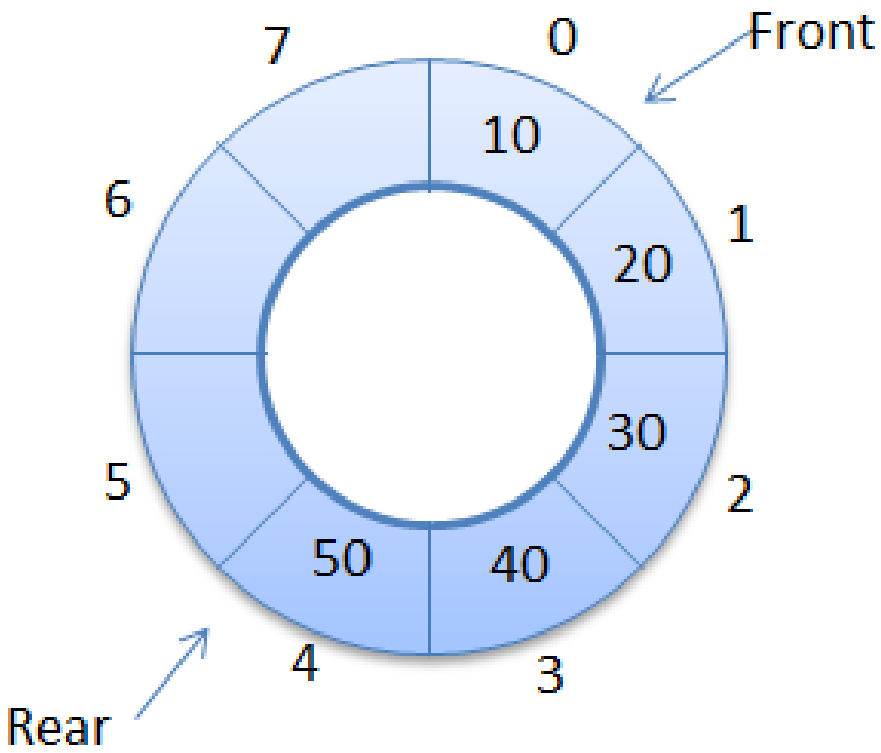
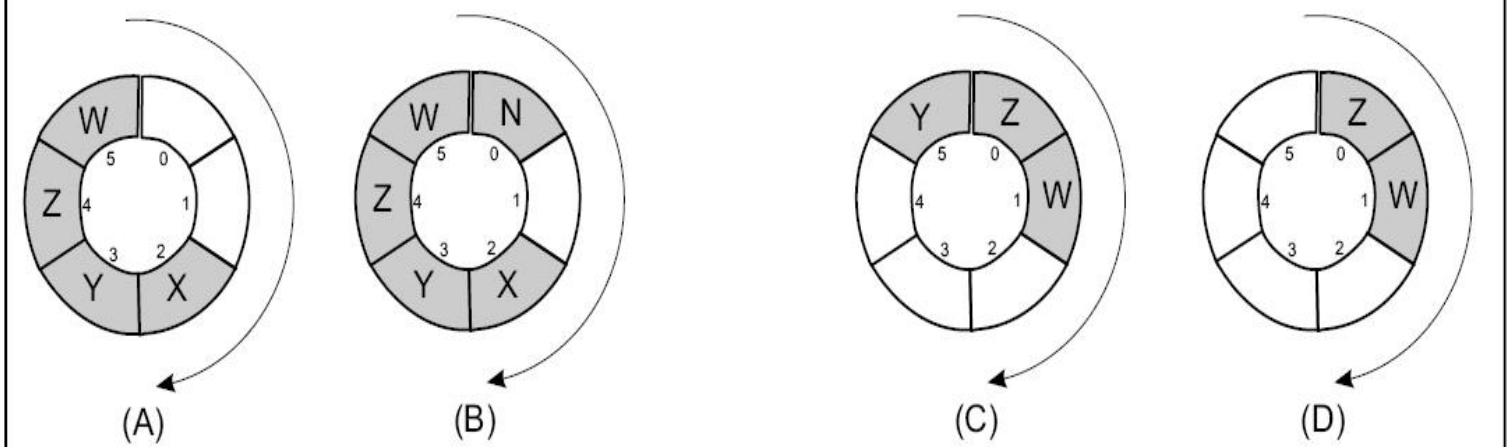


Figura 6: Notação aqui empregada

Vetor Circular: Explicações

	Frente	Cauda	Vetor 0	1	2	3	4	5	Operação
A)	2	5			X	Y	Z	W	Insere N
B)	2	0	N		X	Y	Z	W	

	Frente	Cauda	Vetor 0	1	2	3	4	5	Operação
C)	5	1	Z	W				Y	Retira da Fila
D)	0	1	Z	W					



Função de Criação

- A função que cria uma fila, deve criar e retornar o ponteiro de uma fila vazia
- A função deve informar onde é o início da fila, ou seja, fazer `f->ini = 0`, como podemos ver no código abaixo
- A complexidade de tempo para criar a fila é constante, ou seja, $O(1)$

```
1  /* Aloca dinamicamente a estrutura Fila, inicializando seus
2   * campos e retorna seu ponteiro. A fila depois de criada
3   * estarah vazia.
4   */
5  Fila* criar(void)
6  {
7      Fila* f = malloc(sizeof(Fila));
8      f->n = 0; // quantidade CORRENTE elementos
9      f->ini = 0; // inicio
10     return f;
11 }
```

Função de Inserção

- Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por n.
- Devemos assegurar que há espaço para inserção do novo elemento no vetor, haja vista se tratar de um vetor com capacidade limitada.
- A complexidade de tempo para inserir um elemento na fila é constante, ou seja, $O(1)$.

```
1  /* Insere o elemento e no final da fila f.*/
2  void insere(Fila* f, int e)
3  {
4    int fim;
5    if (f->n == N){
6       printf("Fila cheia!\n"); }
7    else{
8       fim = (f->ini + f->n) % N;
9       f->elementos[fim] = e;
10      f->n++;
11   }
12 }
```

Função de Remoção

- A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno.
- Para remover um elemento, devemos verificar se a fila está ou não vazia.
- A complexidade de tempo para remover um elemento da fila é constante, ou seja, $O(1)$.

```
1  int retira(Fila* f)
2  {
3    int e;
4    if ( vazia(f) )
5      printf("Fila vazia!\n");
6    else{
7      e = f->elementos[f->ini];
8      f->ini = (f->ini + 1) % N;
9      f->n--;
10   }
11   return e;
12 }
```

Exemplo de Uso da Fila

```c
#define N 10
#include <stdio.h>
#include "fila.h"  // TDA.h

int main(void)
{
    Fila * f = criar();

    int i;
    for (i = 0; i < N; i++)
      insere(f, i * 2);

    printf("\nElementos removidos: ");

    for (i = 0; i < N/2; i++)
     printf("%d ", retira(f));

}
```

Onde estamos ...

Fila de Prioridade

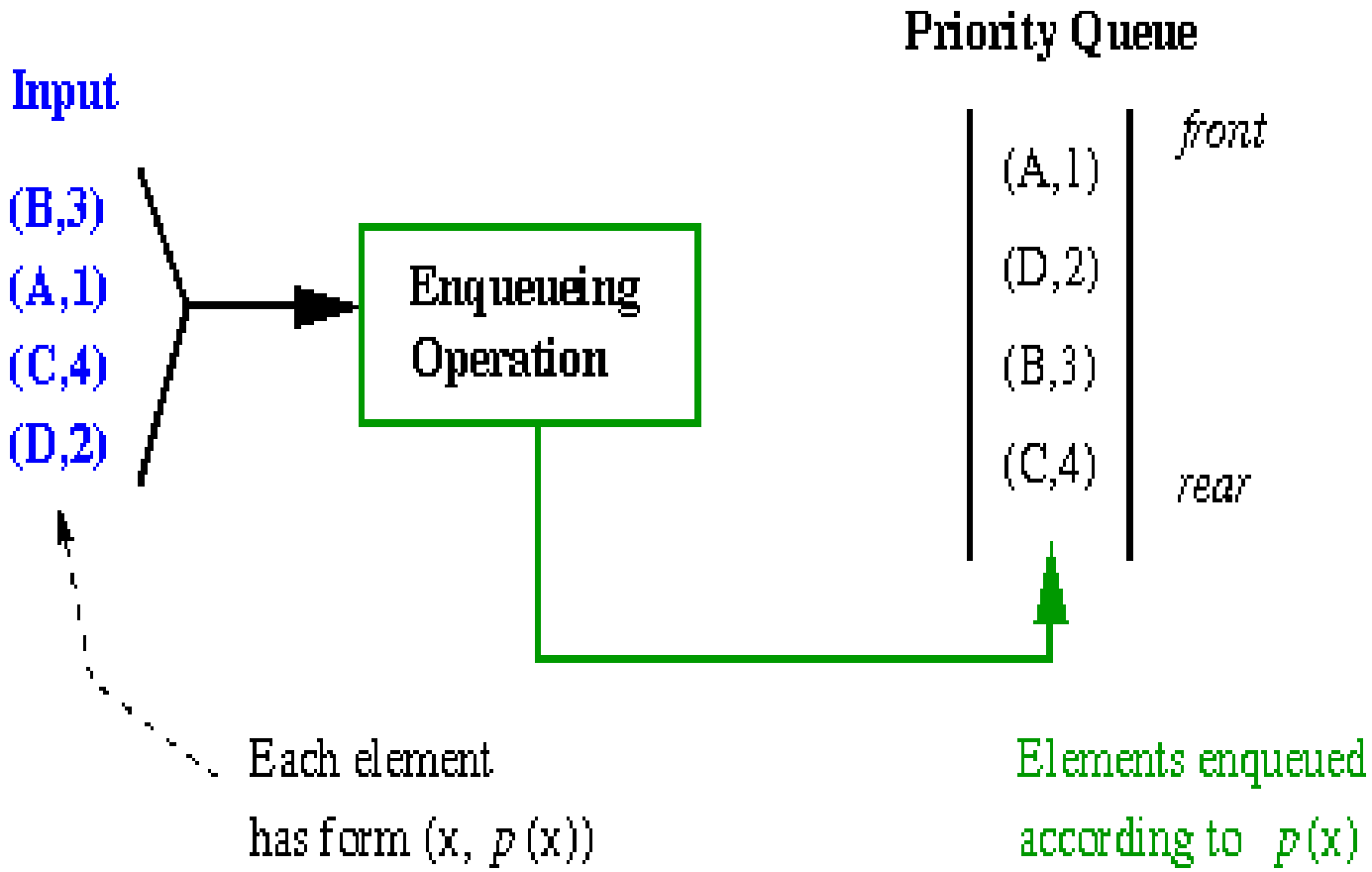


Figura 7: Várias aplicações para um único atendedor

Filas de Prioridade – Várias

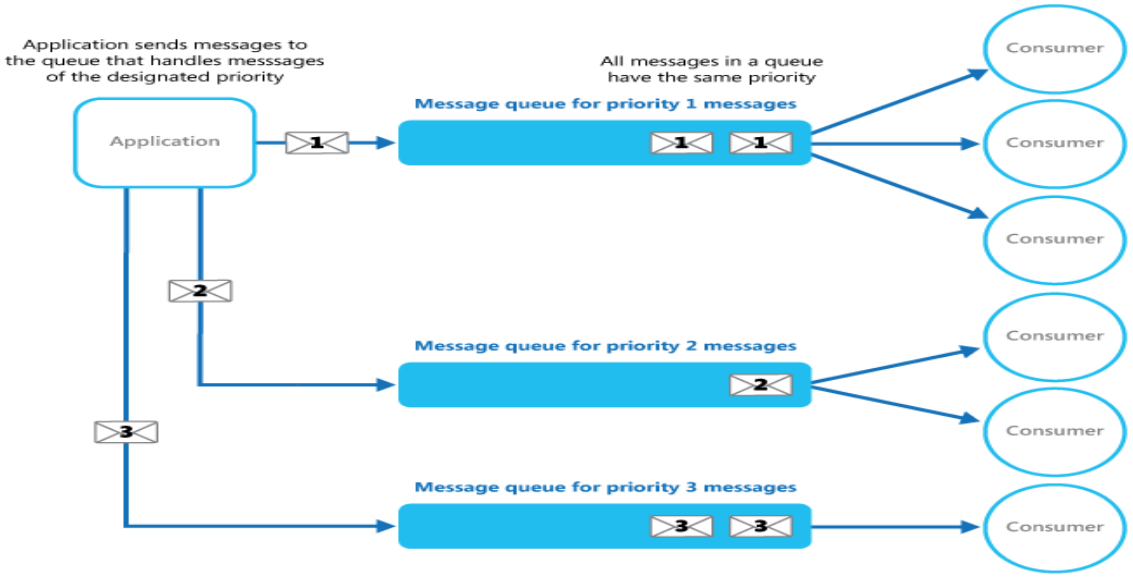


Figura 8: Filas com prioridades e múltiplos atendedores

Filas de Prioridade – Uso de CPU

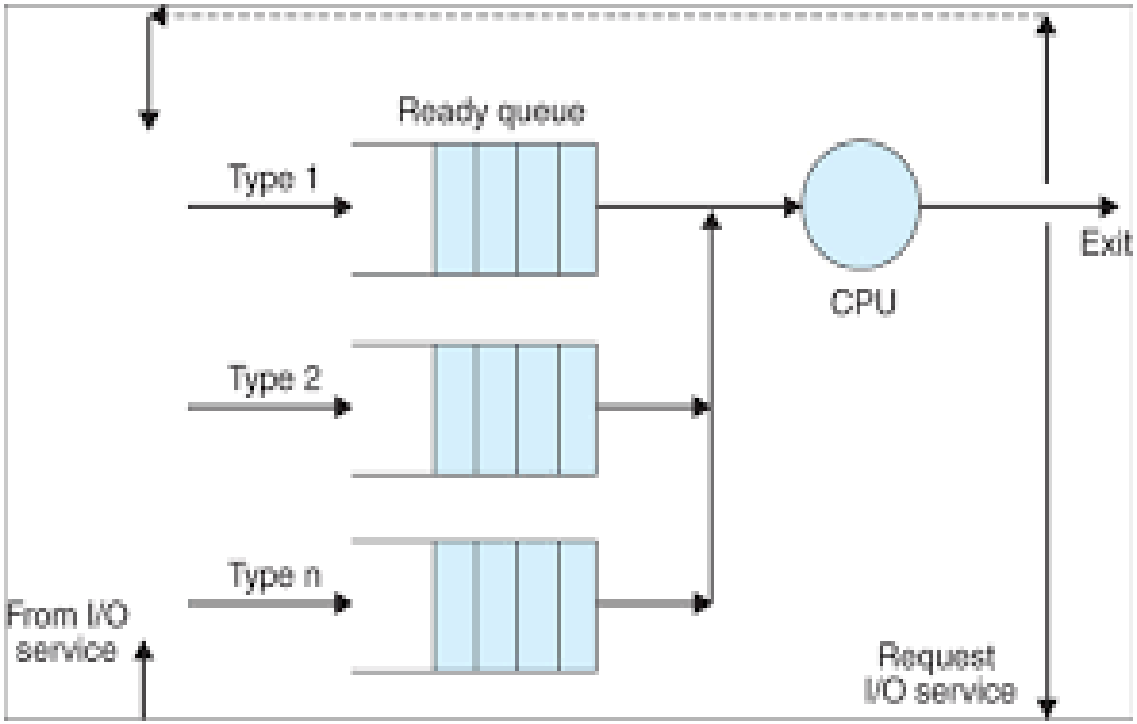


Figura 9: Filas com prioridades – clássico *round-robin*

Filas de Prioridade – Múltiplos Atendedores

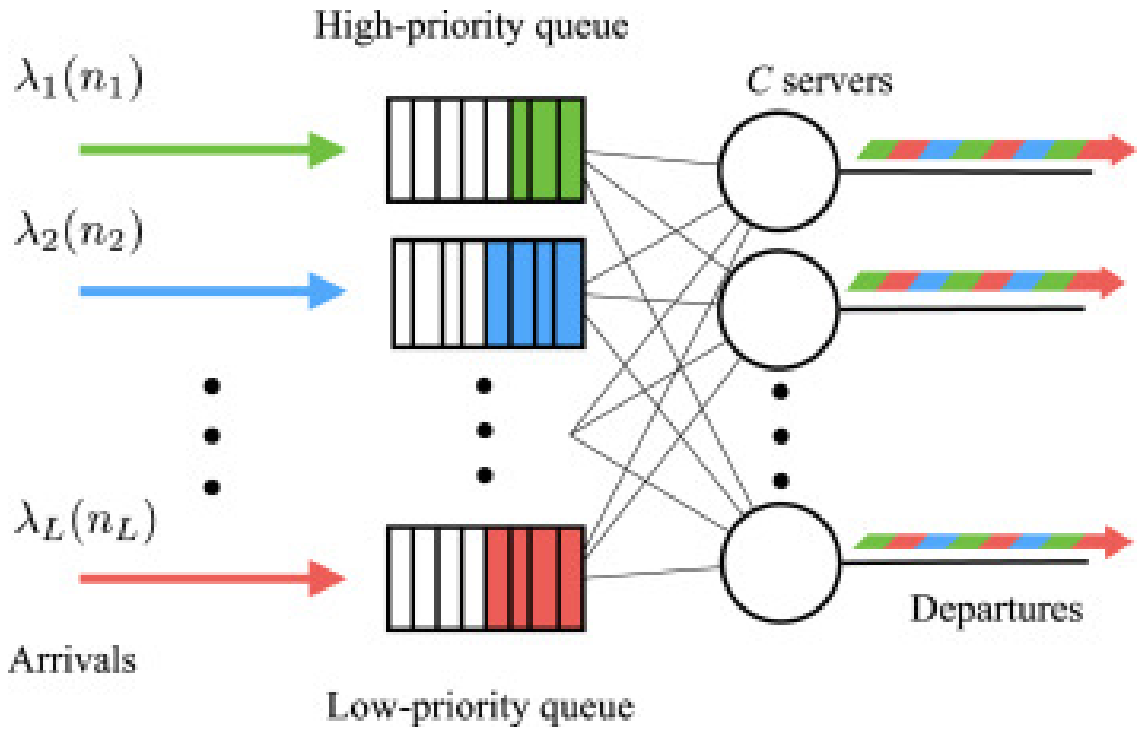


Figura 10: Filas com prioridades – clássico *round-robin*

Filas com Múltiplos Atendedores

BFS: Busca em Largura

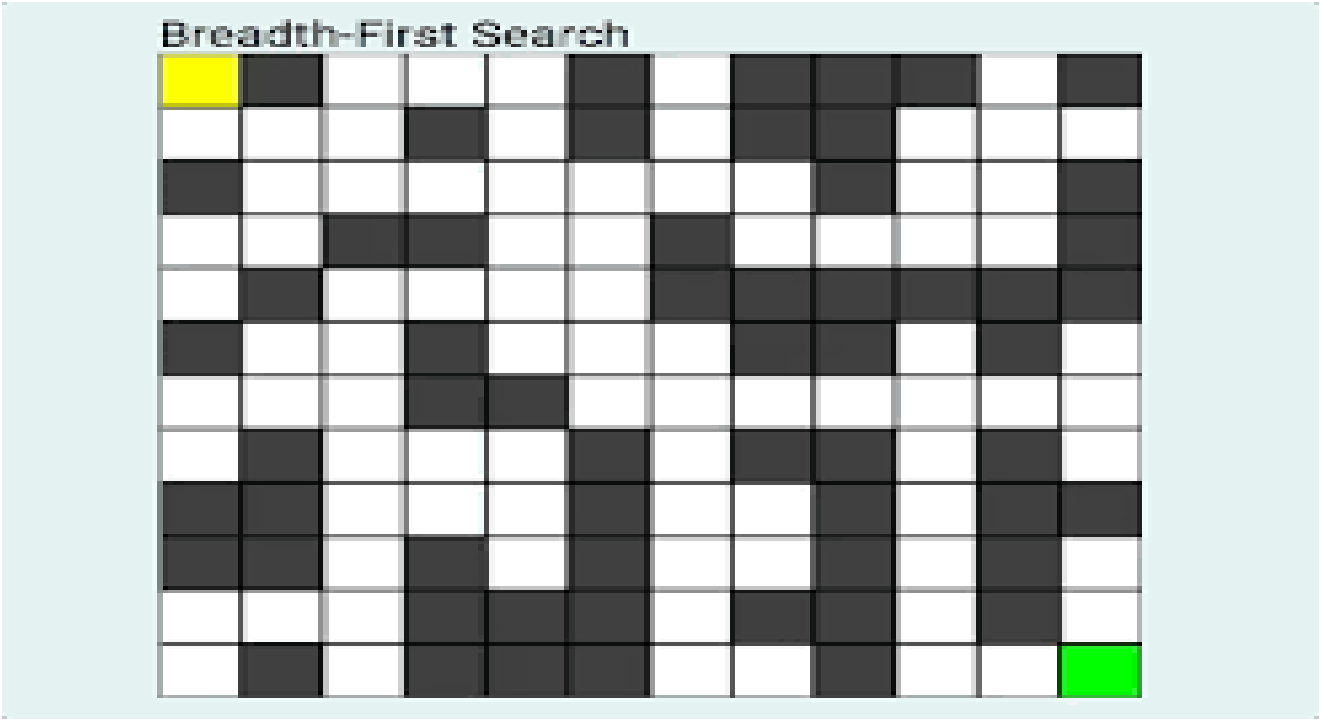


Figura 11: Casos clássicos – pesquisas em labirinto – grafos

Comparativo BFS X DFS *breadth first search (BFS) and depth first search (DFS)*

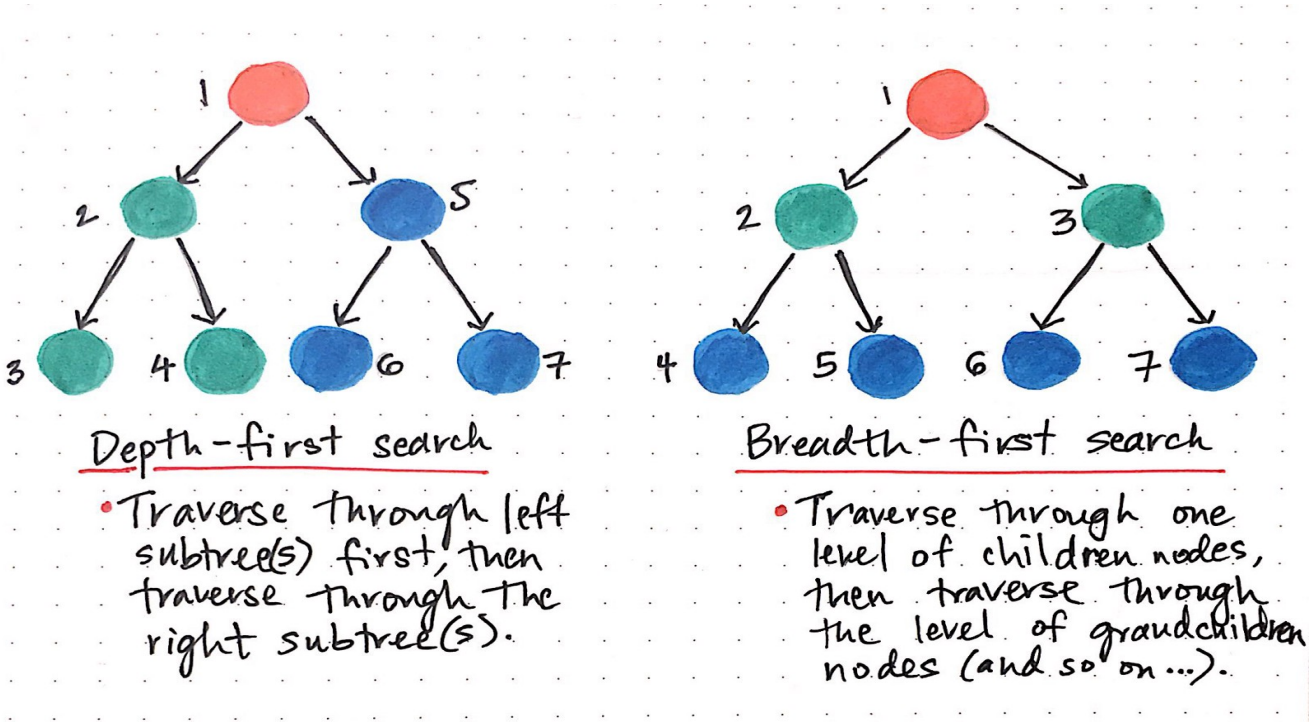


Figura 12: Casos clássicos – pesquisas em labirinto – grafos

BFS *breadth first search (BFS) em Grafo*

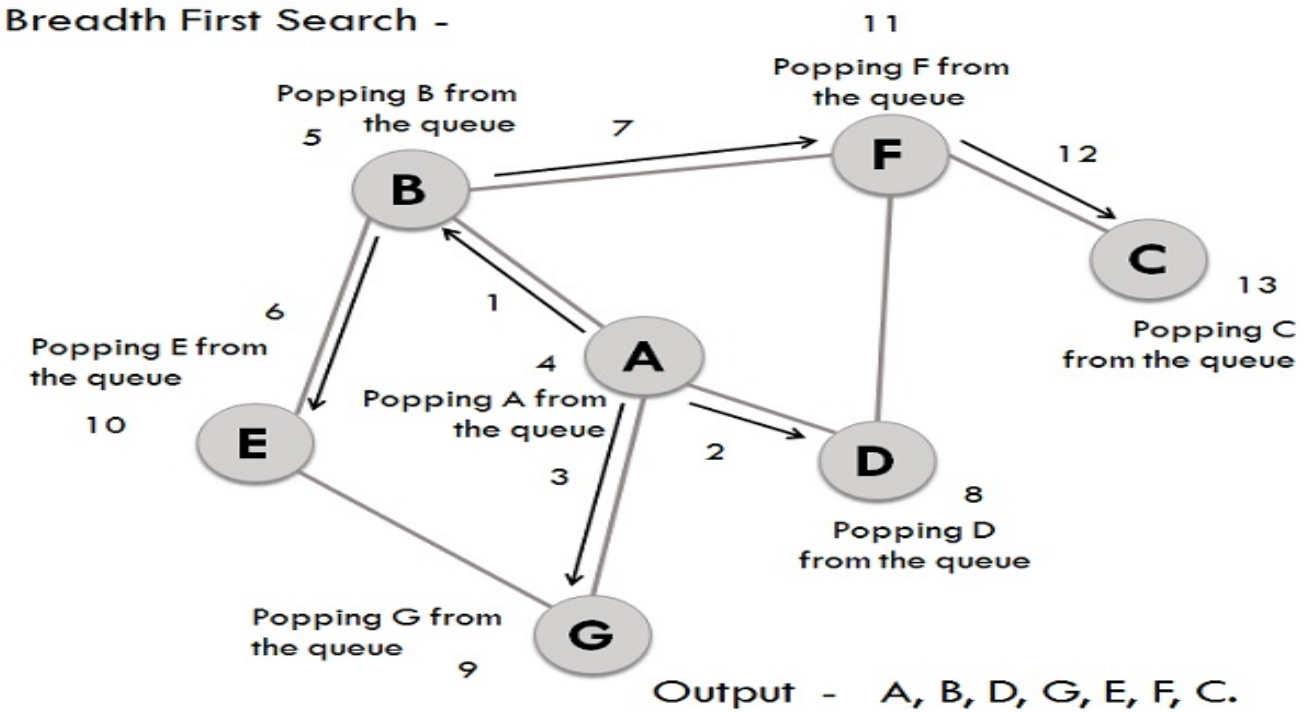


Figura 13: Caso clássico – grafos

A árvore de busca do BFS (*breadth first search*) em Grafos

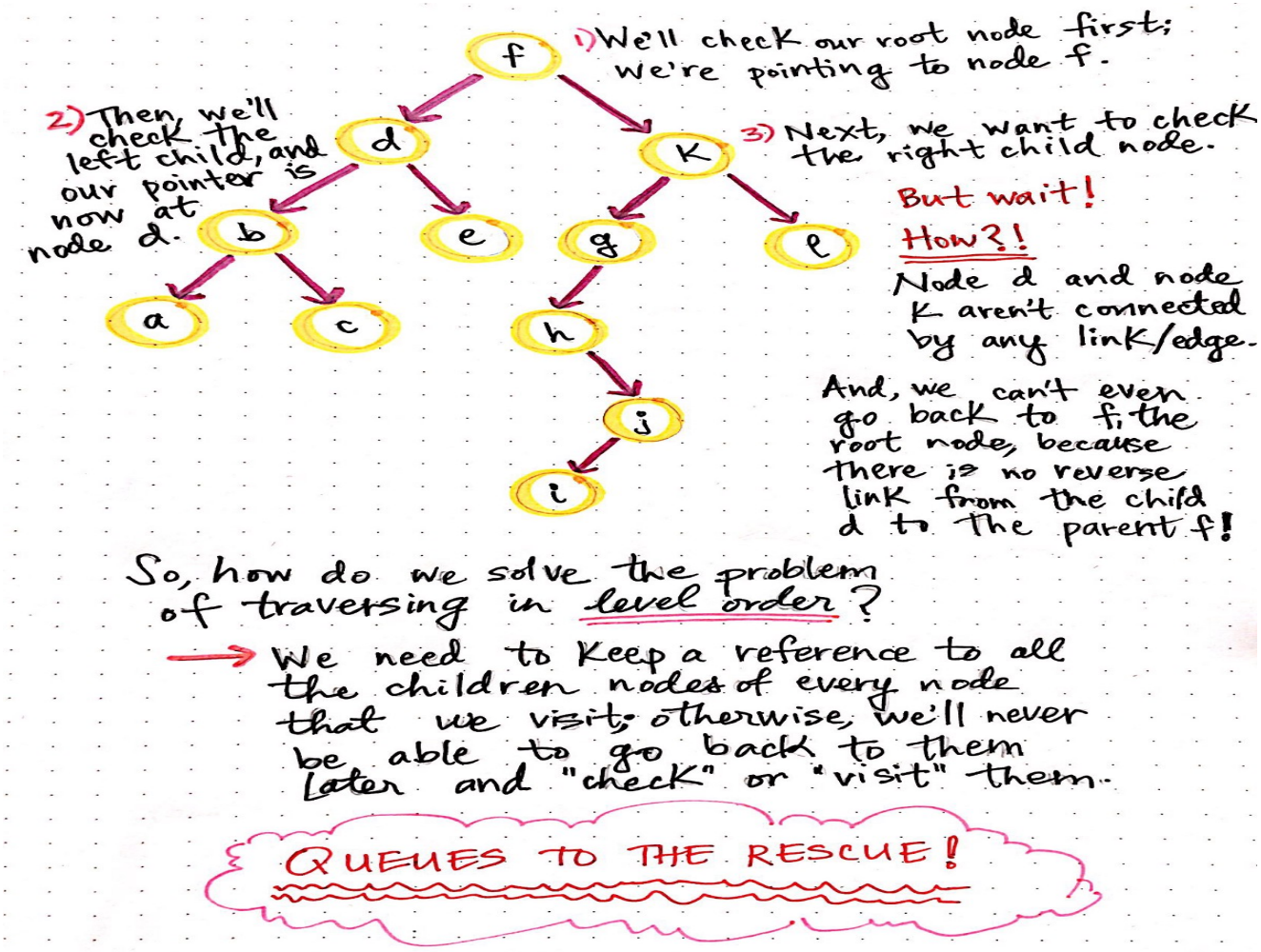


Figura 14: A árvore de busca do BFS

A árvore de busca do BFS (*breadth first search*) em Grafos

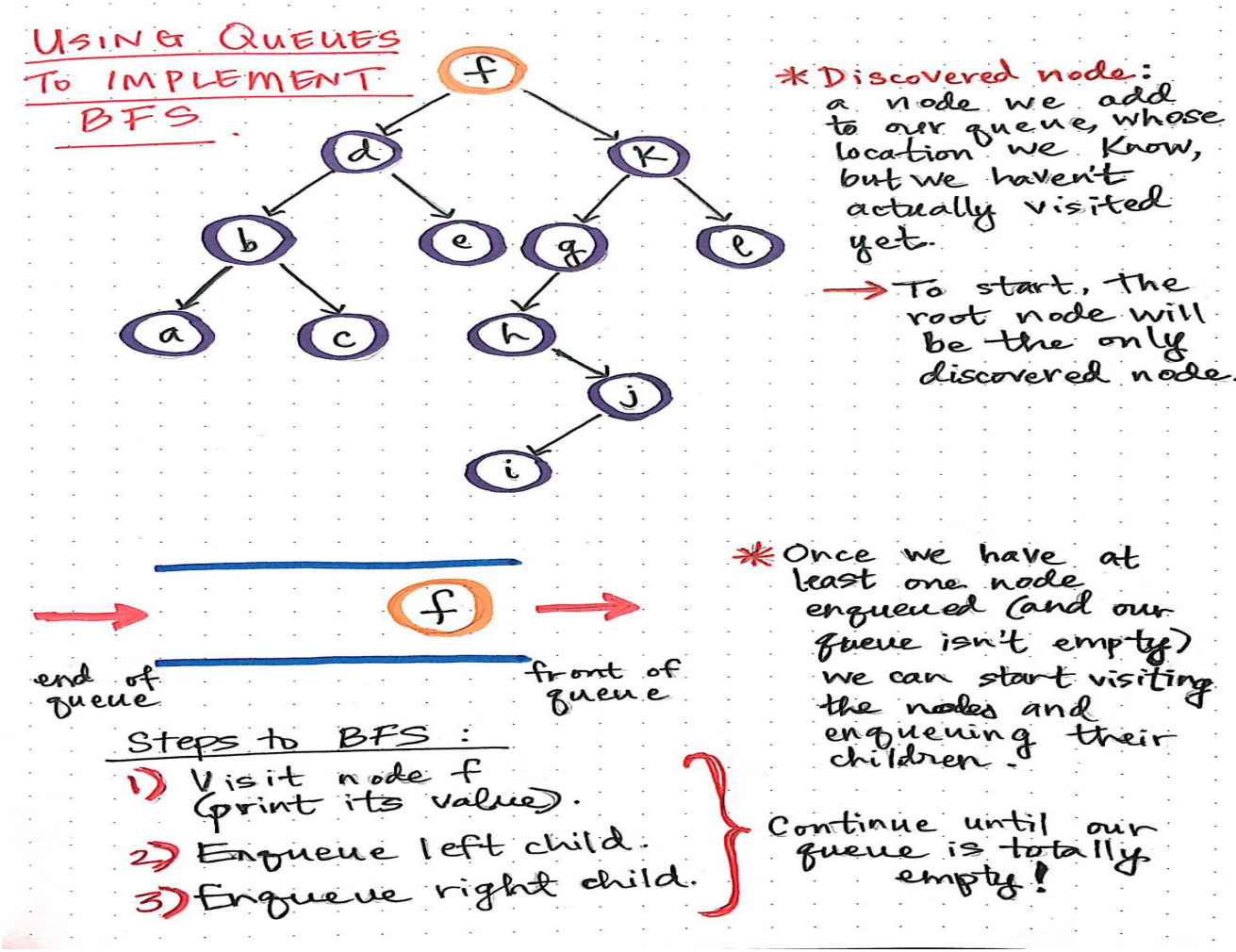


Figura 15: A árvore de busca do BFS

A árvore de busca do BFS (*breadth first search*) em Grafos

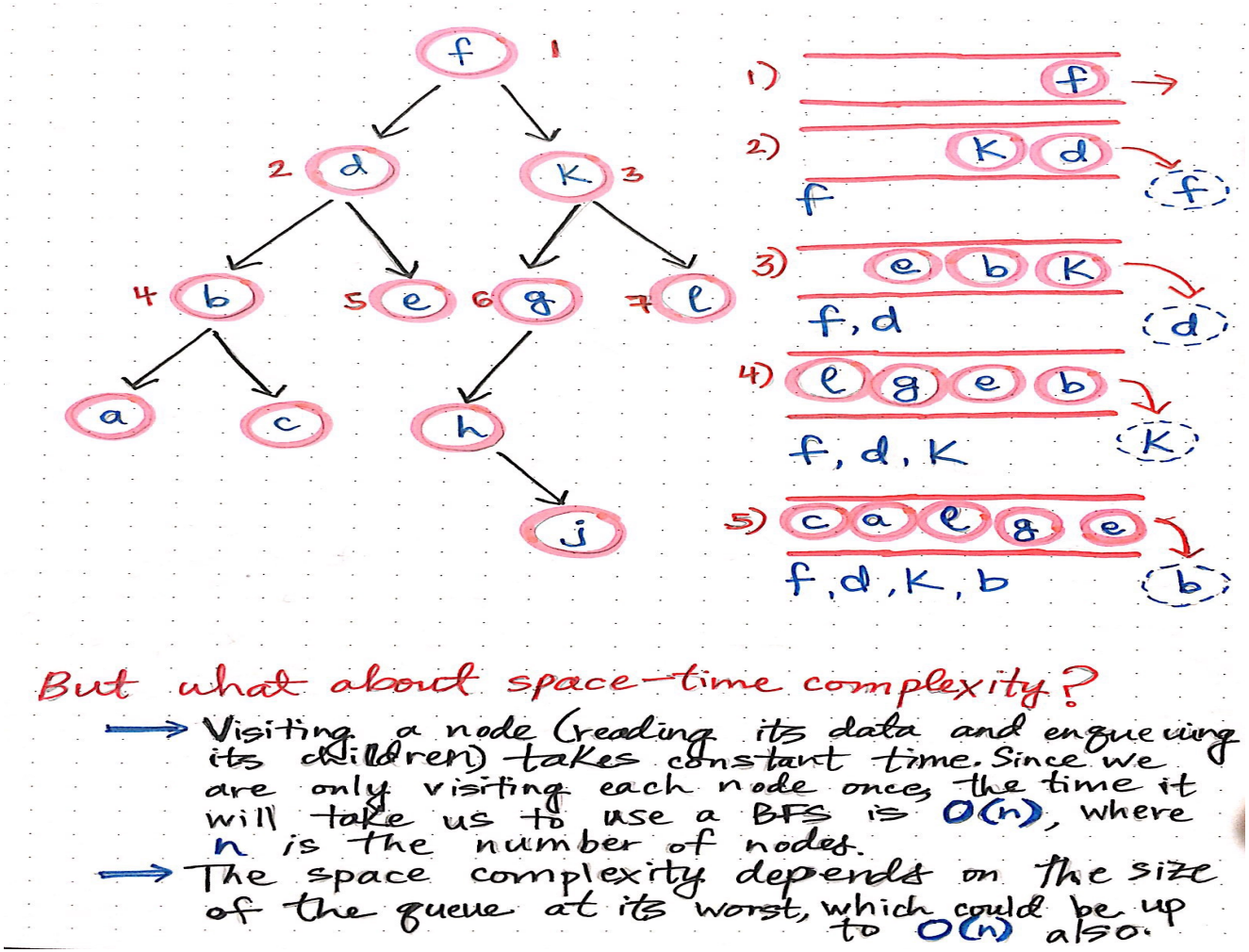


Figura 16: A árvore de busca do BFS

Pseudo Código do BFS (*breadth first search*)

Busca em Largura usando Estrutura de Fila (F)

Pré-requisitos:

- Leitura de uma matriz de adjacência de G e o total de vértices (implícito)
- Uma fila F e seus métodos – chegada (push) e partida (pop)
- Um vetor de status de n-vértices
- Nó de partida ou inicial

```
varredura_do_grafo(G)
{
   fila * F = cria_fila ( MAX ); //cria a fila – estrutura base
   vetor_status_nós[i] = aberto; //todos os i-nós marcados como não visitados
   no_inicial = lê_no_inicial( );
   BFS( no_inicial, F );            // chama BFS com o nó inicial e a FILA F
 }

BFS( vert, F )
{
   chegada(vert, F);     // vertice = nó inicial da busca … só uma vez!
   enquanto ( !vazia(F) )          // enquanto fila não estiver vazia
    {
       x = partida(F);          // obtém um vértice x  a ser inspecionado
       se ( vetor_status_nós[x] == aberto )
         {
           imprime_ou_visita(x);
           teste_o_que_quiseres(x); // AQUI se testa x …
           vetor_status_nós[x] = fechado; // nó visitado, muda status
         };
        para_todos_vértices de i = 0 até i <= vertices  faça
        // aqui muda o sentido da busca … esquerda ou direita
         {
           se (  (vetor_status_nós[i] == aberto ) AND
                 (vertice_vizinho[i] == 1)
           chegada( i, F );
         };  // fim do empilhamento dos nos adjacentes novos
    } ;  // fim do enquanto
 fim_do_BFS;
} ;
```

Resumindo BFS X DFS *breadth first search (BFS) and depth first search (DFS)* em um labirinto

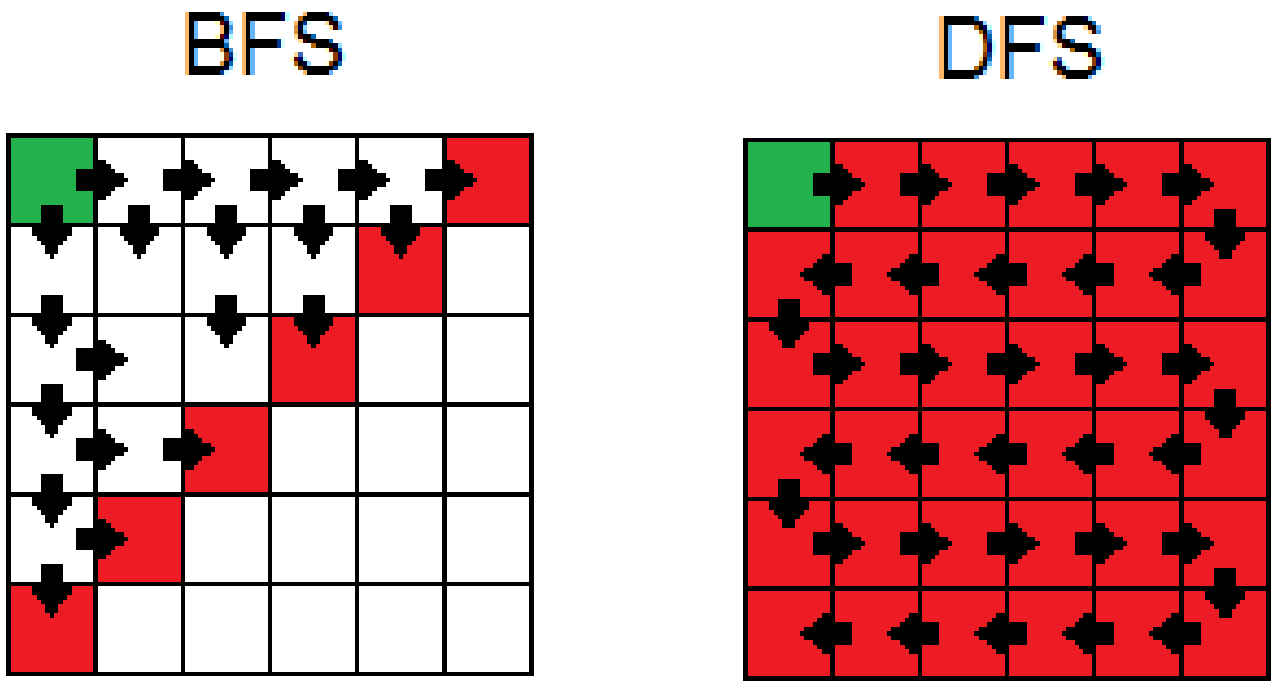


Figura 17: Comportamento no labirinto – célula à esquerda é o final

Representando este labirinto como um grafo

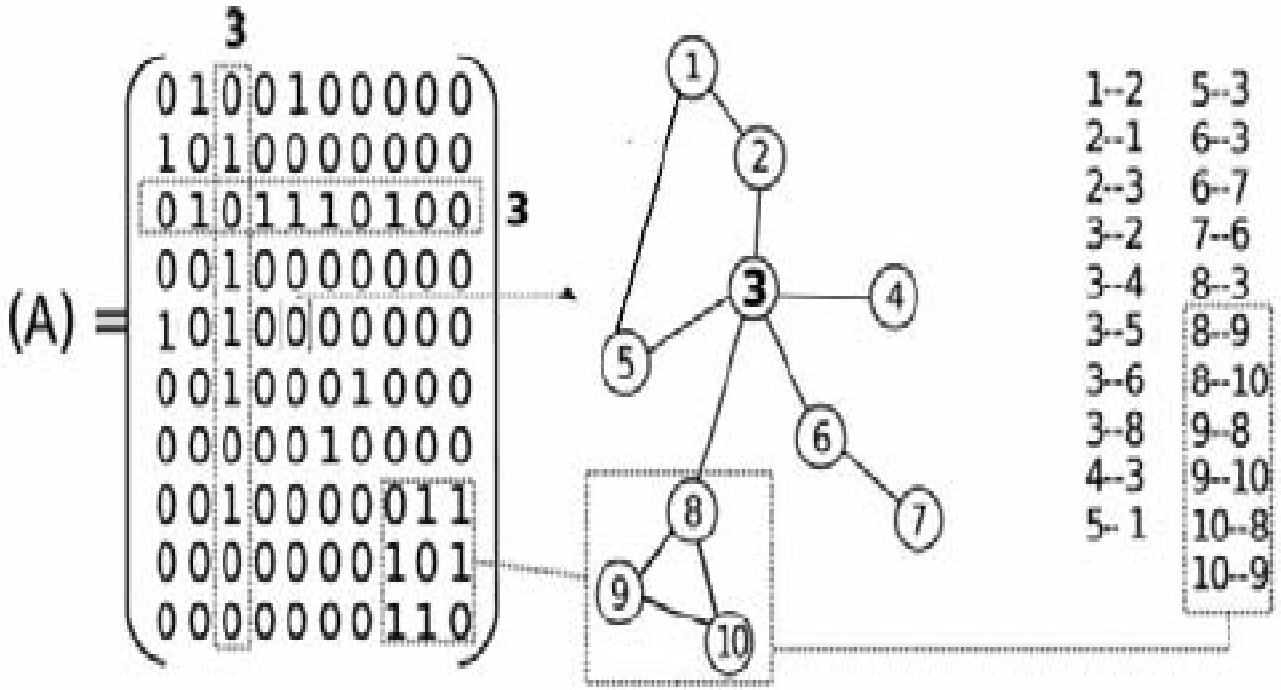


Figura 18: **0**: parede (limite) – **1**: livre para se movimentar

Exercícios

1. Implemente um esquema de uma fila usando duas pilhas
2. Encontrar o maior sub-conjunto de uma janela w para um vetor circular N. Basicamente, preencher o vetor inteiro, posicionar inicio em fim de fila, tal que a diferença seja w. Rotacione com acionamentos de chegadas e partidas, até encontrar a maior soma neste vetor. Ou seja, $|$`rear - front + 1`$| =$ `w` $|$. Função: `abs(rear - front + 1) = w`
3. Construa uma função que copie uma fila, para uma nova fila invertida.
4. Faça uma simulação aleatória de chegadas e partidas em uma fila, tal que a cada passo exiba as mensagens de fila cheia, fila vazia, número de elementos corrente na fila. Faça isto para um número considerável de simulações. Este exercício é a base na simulação de sistemas de filas.
5. Implemente um sistema de fila com prioridades (será um dos projetos)

Referências

1. Karumanchi, Narashimha (2017). *Data Structures and Algorithms Made Easy – Data Structures and Algorithms and Puzzles.* CareerMonk.com
2. Tenenbaum, A. M., Langsam, Y., and Augestein, M. J. (1995). *Estruturas de Dados Usando C.* MAKRON Books, pp. 207-250.
3. Wirth, N. (1989). *Algoritmos e Estrutura de Dados.* LTC, pp. 151-165.