Pilhas e Filas

Algoritmos e Estruturas de Dados 2
2017-1
Flavio Figueiredo (http://flaviovdf.github.io)

Pilhas

Pilhas (Stack)

- Estrutura similar às listas lineares que vimos na última aula

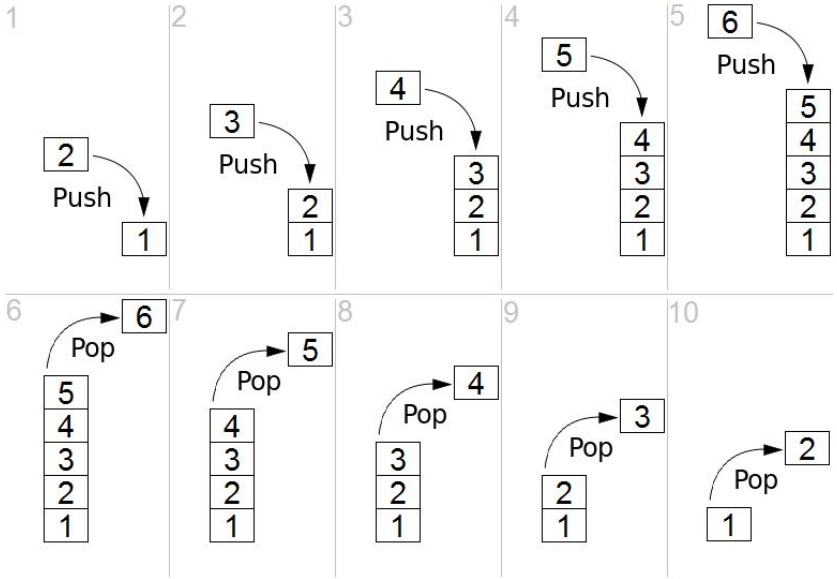
- [Mais Simples] Três operações principais
  - ***stackPush***
    - Insere um elemento no topo da pilha
  - ***stackPop***
    - Remove o elemento do topo da pilha
  - ***stackIsEmpty***
    - Indica se existe algum elemento na pilha

Pilhas (Stack)

- Estrutura similar às listas lineares que vimos na última aula

- [Mais Simples] Três operações principais
  - ***stackPush***
    - Insere um elemento no topo da pilha
  - ***stackPop***
    - Remove o elemento do topo da pilha
  - ***stackIsEmpty***
    - Indica se existe algum elemento na pilha

- Uma pilha de pratos para limpar
- Uma pilha de provas para corrigir



[Calma] Eu já aprendi a lista linear, a mesma consegue remove e inserir do início.
Preciso saber sobre pilhas?

TADs São Contratos

- Se você precisa de uma pilha, melhor assinar o contrato da mesma
- Podemos implementar tudo sem tads, com vetores etc
- Boas práticas de programação usam os TADs certos nos momentos certos

Contrato da Pilha

- As funções que precisamos são estas
- Mais de uma forma de implementar
- Usar Vetores por baixo
  - Similar ao que vimos na aula passada
- Usar a Lista Encadeada por baixo
- Vamos fazer com ponteiros!

```c
#ifndef STACK_H
#define STACK_H

stack_t *stackCreate();
void stackPush(stack_t *stack, int value);
int stackPop(stack_t *stack);
int stackIsEmpty(stack_t *stack);
void stackFree(stack_t *stack);

#endif
```

Métodos

- stackCreate
  - Cria a Pilha, aloca memória
- stackPush
  - Insere um elemento no **topo** da Pilha
- stackPop
  - Remove um elemento do **topo** da pilha
- stackIsEmpty
  - Indica se a pilha está vazia
- stackFree
  - Libera a memória

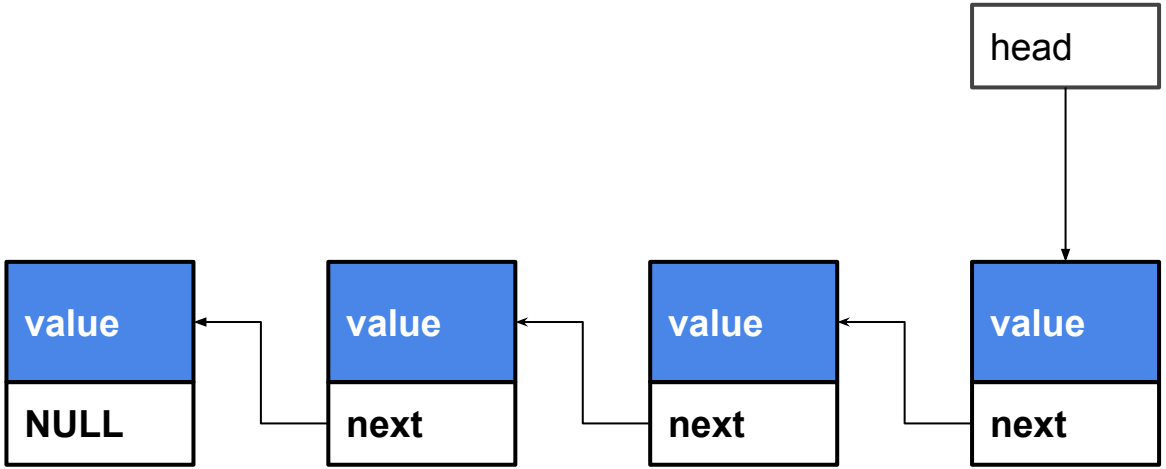
Structs + Apontadores

- Adicionando os structs
- stack_node_t
  - Similar a lista encadeada
  - Apontamos apenas para o próximo elemento
- stack_t
  - Um apontador para o topo da pilha
- Lembre-se
  - Value em int por simplicidade na aula

```c
#ifndef STACK_H
#define STACK_H

typedef struct stack_node {
  int value;
  struct stack_node *next;
} stack_node_t;

typedef struct {
  stack_node_t *head;
} stack_t;

stack_t *stackCreate();
void stackPush(stack_t *stack, int value);
int stackPop(stack_t *stack);
int stackIsEmpty(stack_t *stack);
void stackFree(stack_t *stack);

#endif
```

Abstraindo

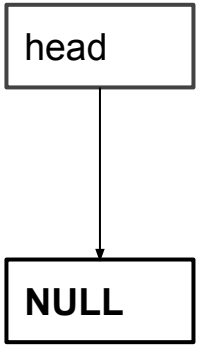
```c
typedef struct {
  stack_node_t *head;
} stack_t;
```



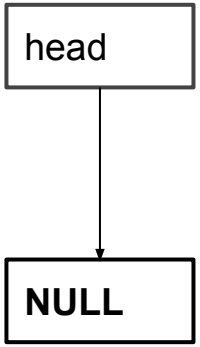

```c
typedef struct stack_node {
  int value;
  struct stack_node *next;
} stack_node_t;
```

Implementando: stackCreate

stackCreate

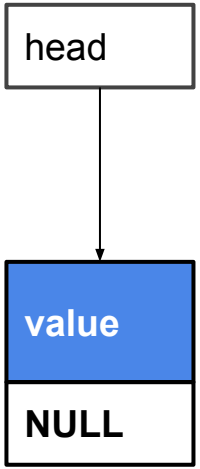


stackCreate

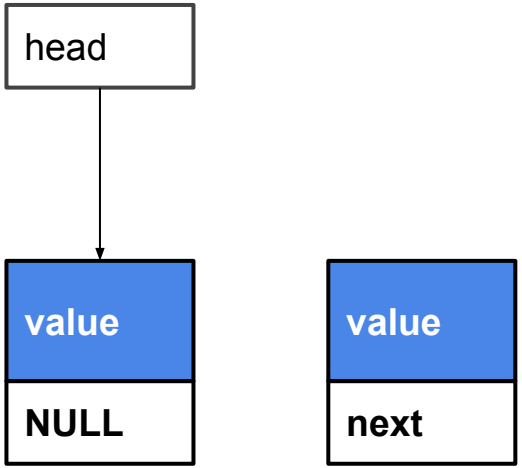



```
stack_t *stackCreate() {
  stack_t *stack = (stack_t *) malloc(sizeof(stack_t));
  if (stack == NULL) {
    printf("Memory error");
    exit(1);
  }
  stack->head = NULL;
  return stack;
}
```

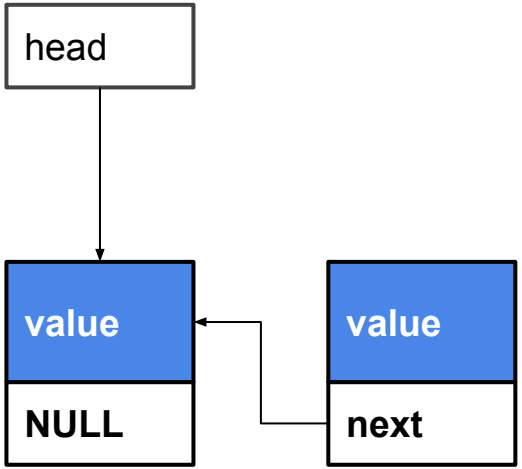
stackPush



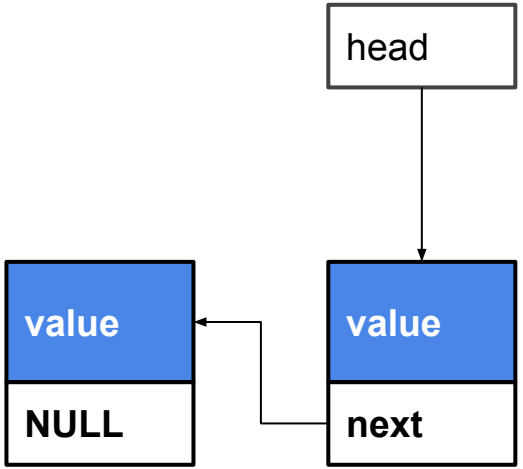
stackPush



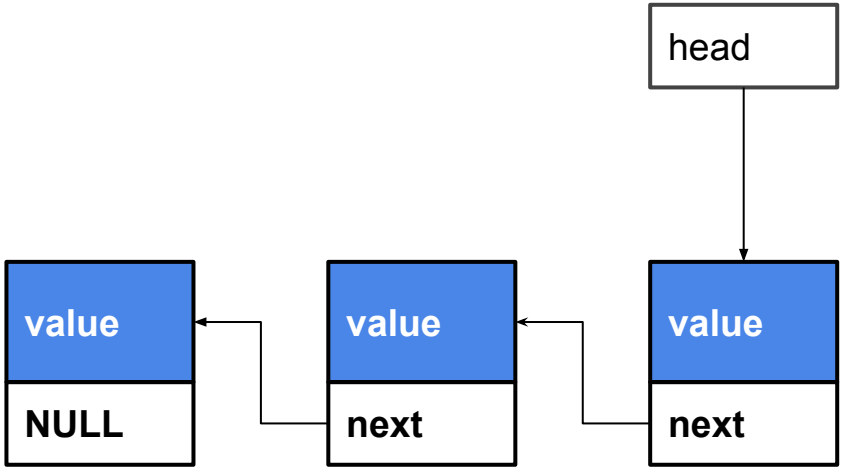
stackPush



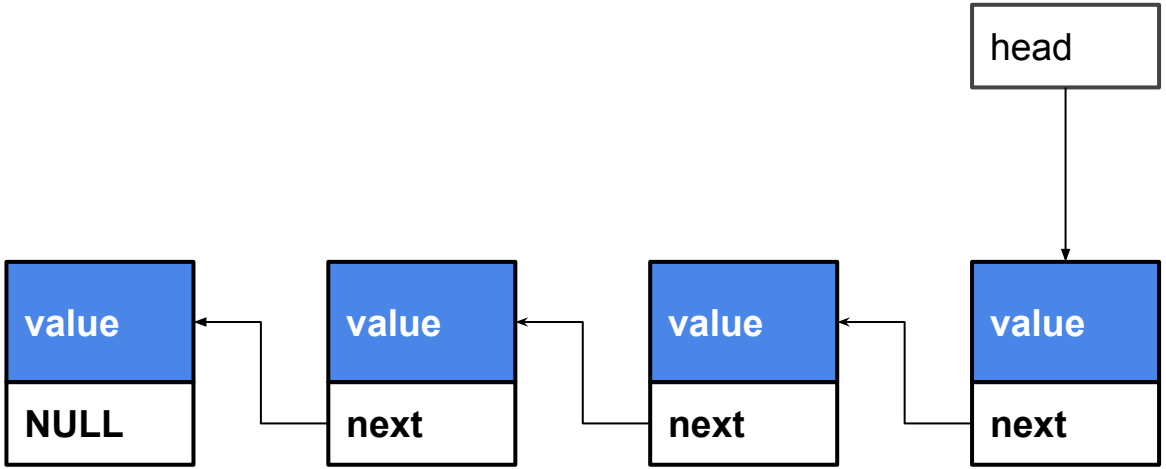
stackPush



stackPush

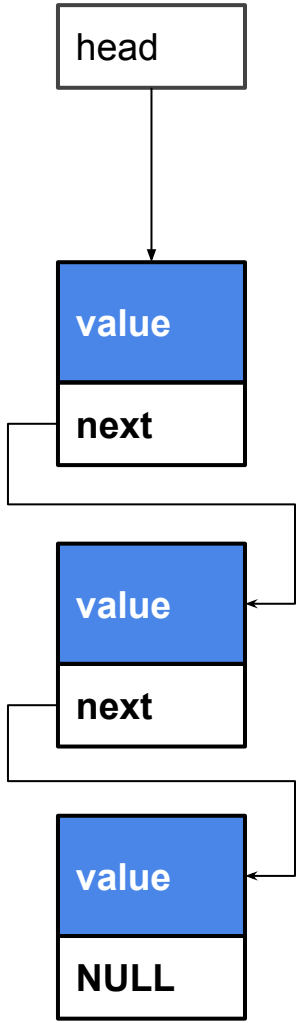


stackPush (algumas chamadas depois)



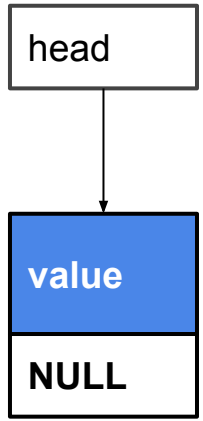
stackPush

- Observe como head sempre aponta para o última elemento
- O primeiro sempre tem NULL como next
- O efeito é como a pilha ao lado
  - Apenas vita na horizontal
- Sempre lidamos com o topo da pilha
  - Head



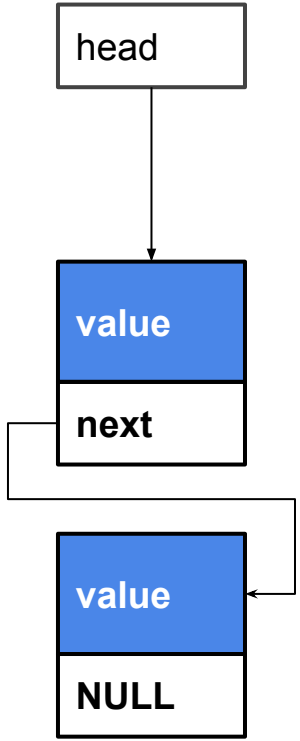
stackPush

```
void stackPush(stack_t *stack, int value) {
  stack_node_t *node = (stack_node_t *) malloc(sizeof(stack_node_t));
  if (node == NULL) {
     printf("Memory error");
     exit(1);
  }
  node->value = value;
  node->next = stack->head;
  stack->head = node;
}
```



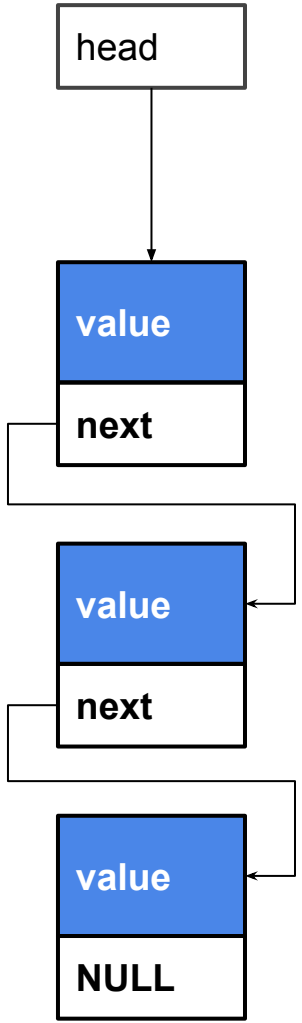
stackPush

```
void stackPush(stack_t *stack, int value) {
  stack_node_t *node = (stack_node_t *) malloc(sizeof(stack_node_t));
  if (node == NULL) {
     printf("Memory error");
     exit(1);
  }
  node->value = value;
  node->next = stack->head;
  stack->head = node;
}
```

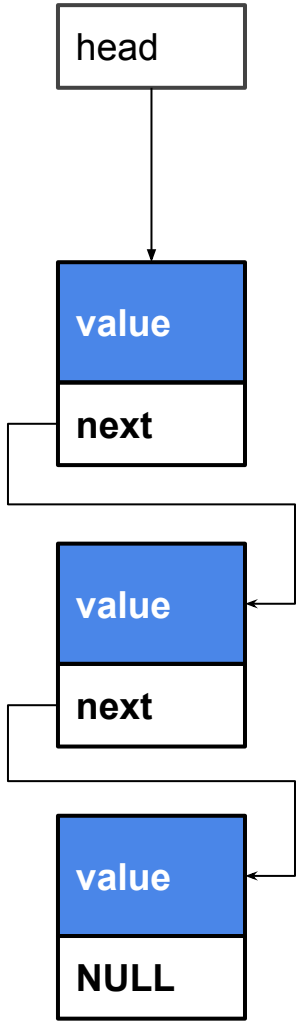


stackPush

```
void stackPush(stack_t *stack, int value) {
  stack_node_t *node = (stack_node_t *) malloc(sizeof(stack_node_t));
  if (node == NULL) {
     printf("Memory error");
     exit(1);
  }
  node->value = value;
  node->next = stack->head;
  stack->head = node;
}
```

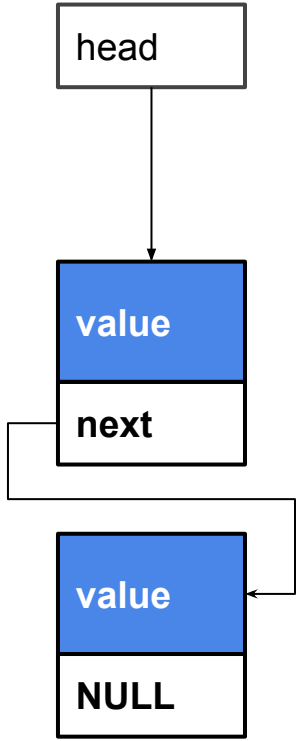


stackPop



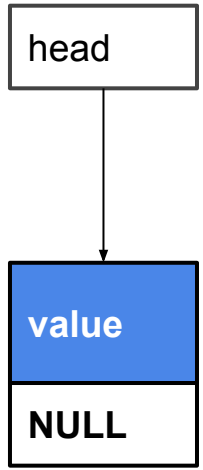
stackPop

- Reduzimos o tamanho da Pilha
- Tiramos um "prato" do topo
  - 1 prova
  - 1 artigo para ler
- Isto tem alguma relação com chamada de funções?



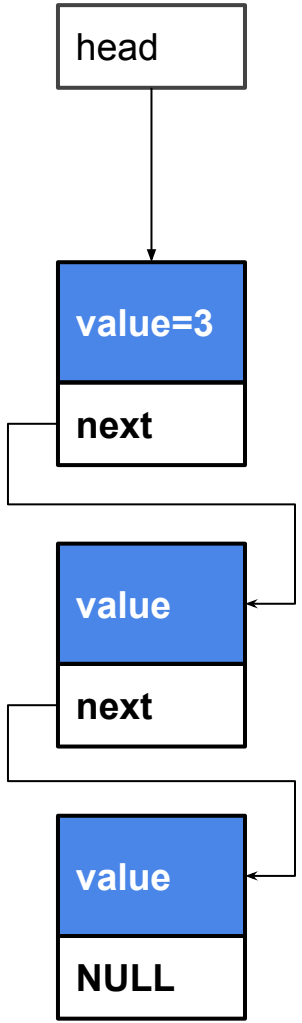
stackPop

- Reduzimos o tamanho da Pilha
- Tiramos um "prato" do topo
  - 1 prova
  - 1 artigo para ler
- Isto tem alguma relação com chamada de funções?
  - Quando chamamos uma função empilhamos a mesma
  - Fica no topo, quem executa agora
  - Ao remover, voltamos para a anterior
  - Por isso dizemos que a chamada de funções é uma Pilha



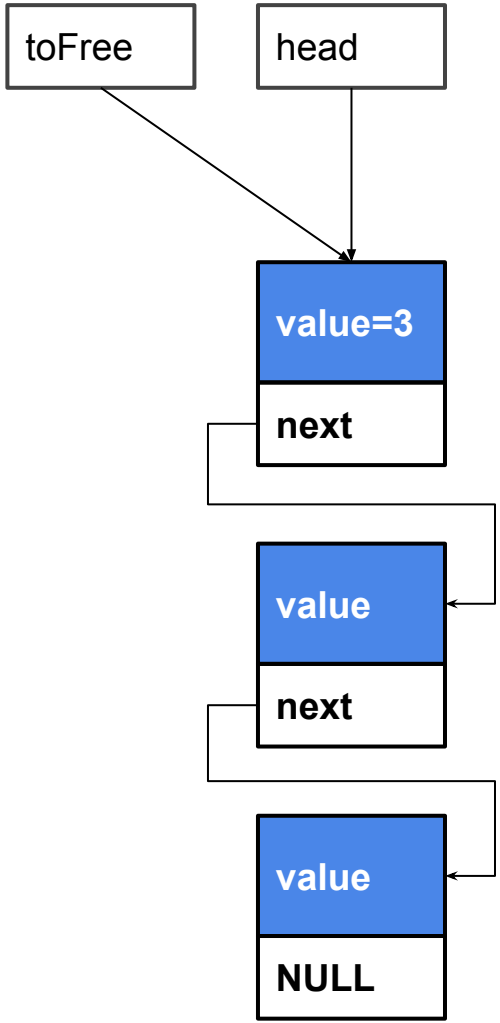
stackPop

```
int stackPop(stack_t *stack) {
  stack_node_t *toFree;
  int toReturn;
  if (stack->head != NULL) {
     toFree = stack->head;
     toReturn = toFree->value;
     stack->head = stack->head->next;
     free(toFree);
  } else {
     printf("Stack is empty!!!");
     exit(1);
  }
  return toReturn;
}
```



stackPop

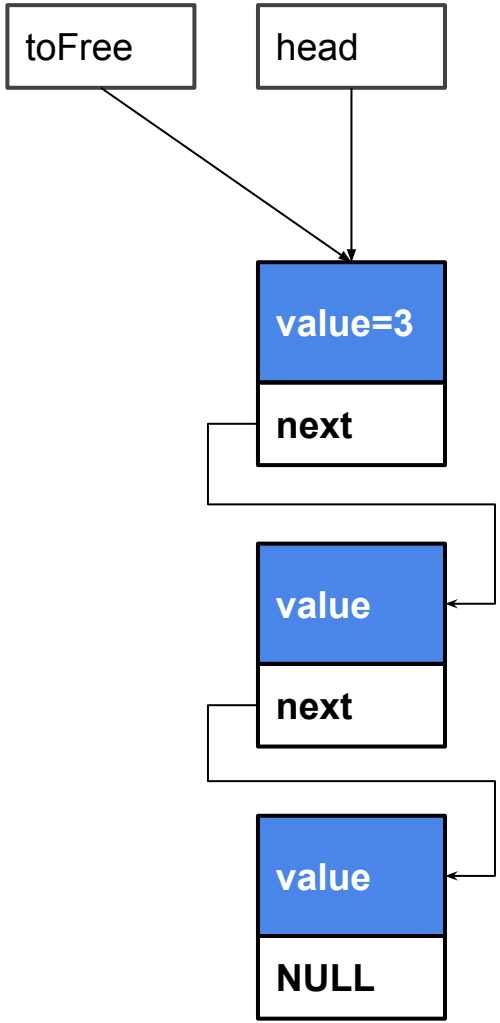
```
int stackPop(stack_t *stack) {
  stack_node_t *toFree;
  int toReturn;
  if (stack->head != NULL) {
     toFree = stack->head;
     toReturn = toFree->value;
     stack->head = stack->head->next;
     free(toFree);
  } else {
     printf("Stack is empty!!!");
     exit(1);
  }
  return toReturn;
}
```



stackPop

```
int stackPop(stack_t *stack) {
  stack_node_t *toFree;
  int toReturn;
  if (stack->head != NULL) {
     toFree = stack->head;
     toReturn = toFree->value;
     stack->head = stack->head->next;
     free(toFree);
  } else {
     printf("Stack is empty!!!");
     exit(1);
  }
  return toReturn;
}
```

```
toReturn = 3;
```

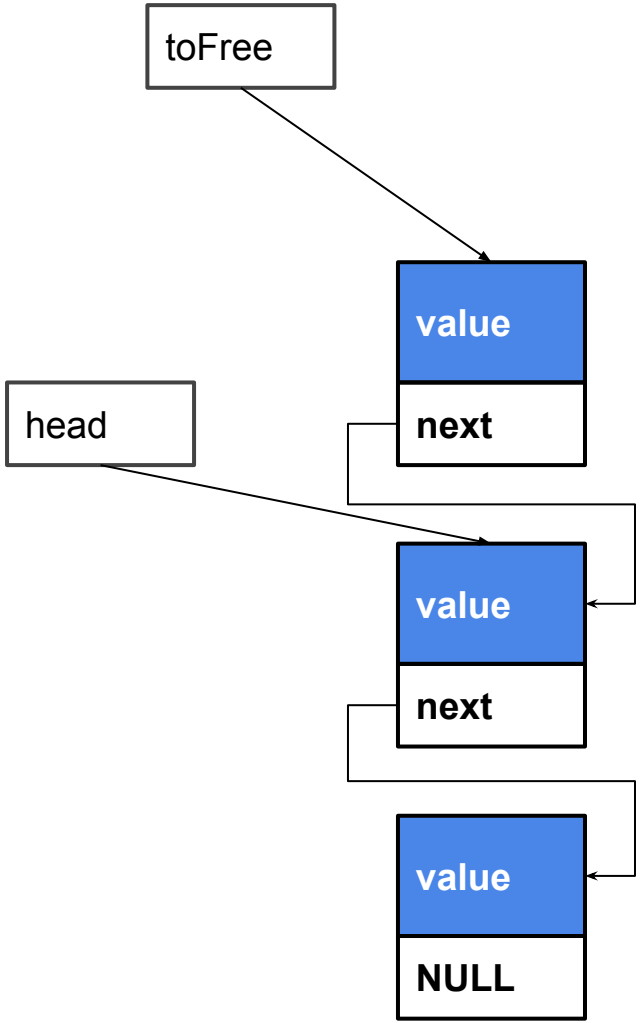


stackPop

```
int stackPop(stack_t *stack) {
  stack_node_t *toFree;
  int toReturn;
  if (stack->head != NULL) {
     toFree = stack->head;
     toReturn = toFree->value;
     stack->head = stack->head->next;
     free(toFree);
  } else {
     printf("Stack is empty!!!");
     exit(1);
  }
  return toReturn;
}
```

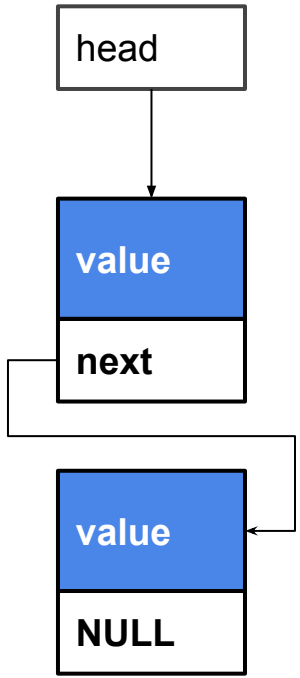
```
toReturn = 3;
```

Atualiza o head



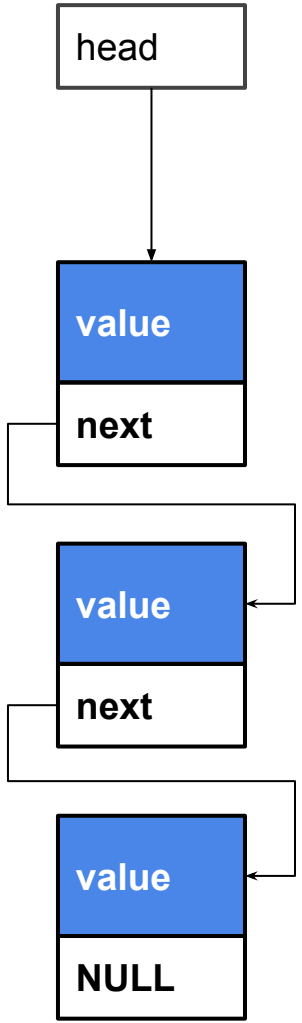
stackPop

```
int stackPop(stack_t *stack) {
  stack_node_t *toFree;
  int toReturn;
  if (stack->head != NULL) {
     toFree = stack->head;
     toReturn = toFree->value;
     stack->head = stack->head->next;
     free(toFree);
  } else {
     printf("Stack is empty!!!");
     exit(1);
  }
  return toReturn;     toReturn = 3;
}
```



stackFree

```
void stackFree(stack_t *stack) {
  stack_node_t *next = stack->head;
  stack_node_t *toFree = NULL;
  while (next != NULL) {
     toFree = next;
     next = next->next;
     free(toFree);
  }
  free(stack);
}
```



stackIsEmpty

```
int stackIsEmpty(stack_t *stack) {
  if (stack->head == NULL) {
     return 1;
  } else {
     return 0;
  }
}
```

Serve para que os usuários do TAD saibam que uma pilha não tem nada!

Um main.c usando o STACK_H

```c
#include <stdio.h>
#include "stack.h"

int main() {
  stack_t *stack = stackCreate();
  int i;
  for (i = 0; i < 10; i++) {
     printf("Push %d\n", i);
     stackPush(stack, i);
  }
  printf("\n");
  while (stackIsEmpty(stack) != 1) {
     printf("Pop %d\n", stackPop(stack));
  }
  for (i = 0; i < 10; i++) {
     stackPush(stack, i * 200);
  }
  stackDestroy(stack);
  return 0;
}
```

```
Push  0
Push  1
Push  2
Push  3
Push  4
Push  5
Push  6
Push  7
Push  8
Push  9

Pop 9
Pop 8
Pop 7
Pop 6
Pop 5
Pop 4
Pop 3
Pop 2
Pop 1
Pop 0
```

```c
#include <stdio.h>
#include "stack.h"

int main() {
  stack_t *stack = stackCreate();
  int i;
  for (i = 0; i < 10; i++) {
     printf("Push %d\n", i);
     stackPush(stack, i);
  }
  printf("\n");
  while (stackIsEmpty(stack) != 1) {
     printf("Pop %d\n", stackPop(stack));
  }
  for (i = 0; i < 10; i++) {
     stackPush(stack, i * 200);
  }
  stackDestroy(stack);
  return 0;
}
```

Pilhas

- Último a entrar é o primeiro a sair
- LIFO
  - Last-in-first-out

Notação Polonesa Reversa

```
3 4 +
7
```

```
5 1 2 + 4 × + 3 −
??
```

https://en.wikipedia.org/wiki/Reverse_Polish_notation

5 1 2 + 4 × + 3 −


HEWLETT PACKARD 48G
32K RAM
RAD POLAR CHARS MODES MEMORY STACK PREV MENU
MTH PRG CST VAR NXT
UP HOME DEF RCL →NUM UNDO PICTURE VIEW SWAP
STO EVAL
ASIN ∂ ACOS ∫ ATAN Σ LOG LN
SIN COS TAN
EQUATION MATRIX EDIT CMD PURG ARG CLEAR DROP
ENTER +/− EEX DEL
USER ENTRY SOLVE PLOT SYMBOLIC
TIME STAT UNITS
I/O LIBRARY EQ LIB
CONT OFF
ON
CANCEL
SPC

1 2 + 4 × + 3 −

Lê 5
stackPush(stack, 5);

5

2 + 4 × + 3 −

Lê 1
stackPush(stack, 1);

1
5

+ 4 × + 3 −

Lê 2
stackPush(stack, 2);

2
1
5

4 × + 3 −

Lê +
E agora?

2
1
5

4 × + 3 −

Lê +
result = stackPop(stack) **+** stackPop(stack);
result = 1 + 2 = 3; //Note que o segundo elemento vem antes

2
1
5

4 × + 3 −

Lê +
result = stackPop(stack) **+** stackPop(stack);
result = 1 + 2 = 3;

1
5

4 × + 3 −

Lê +
result = stackPop(stack) **+** stackPop(stack);
result = 1 + 2 = 3 ;

5

4 × + 3 −

Coloca result na pilha

3
5

× + 3 −

Lê 4
stackPush(stack, 4);

4
3
5

+ 3 –

Lê x
result += stackPop(stack) * stackPop(stack);
result = 3 * 4 = 12;

4
3
5

+ 3 –

Lê x
result += stackPop(stack) * stackPop(stack);
result = 3 * 4 = 12;

5

+ 3 –

Empilha result

12
5

3 –

12 + 5 agora

12
5

3 –

12 + 5 agora


HEWLETT PACKARD 48G
32K RAM
RAD POLAR CHARS MODES MEMORY STACK PREV MENU
MTH PRG CST VAR NXT
UP HOME DEF RCL →NUM UNDO PICTURE VIEW SWAP
STO EVAL
ASIN ∂ ACOS ∫ ATAN Σ 10^x LOG e^x LN
SIN COS TAN 1/x
EQUATION MATRIX EDIT CMD PURG ARG CLEAR DROP
ENTER +/- EEX DEL
USER ENTRY SOLVE PLOT SYMBOLIC
7 8 9 ÷
TIME STAT UNITS
4 5 6 ×
I/O LIBRARY EQ LIB
1 2 3 −
CONT OFF
ON CANCEL
0 SPC +

3 –

12 + 5 agora

17

–

Lê 3

3
17


HEWLETT PACKARD 48G
32K RAM
RAD POLAR CHARS MODES MEMORY STACK PREV MENU
MTH PRG CST VAR NXT
UP HOME DEF RCL →NUM UNDO PICTURE VIEW SWAP
STO EVAL
ASIN ACOS ATAN 10^x LOG e^x LN
SIN COS TAN y^x 1/x
EQUATION MATRIX EDIT CMD PURG ARG CLEAR DROP
ENTER +/- EEX DEL
USER ENTRY SOLVE PLOT SYMBOLIC
TIME STAT UNITS
I/O LIBRARY EQ LIB
CONT OFF
ON
CANCEL
SPC

–

Resultado final é?

3
17

Resultado final é?
17 - 3 = 14;

3
17

5 1 2 + 4 × + 3 −

14

Exercício

- Escreva um programa que lê da entrada uma expressão feita na notação polonesa reversa
- Resolva a expressão
- Use uma pilha
  - Tente implementar do 0 para aprender

Custos

- stackCreate
  - O(1)
- stackPush
  - O(1)
- stackPop
  - O(1)
- stackFree
  - O(n)
- stackIsEmpty
  - O(1)

Exercício

- No TP1

- Como usar uma pilha para imprimir as transações ordenadas por data?
  - Iniciando da mais recente no topo

Filas

Filas (FIFO)

- O “oposto” de uma Pilha

- Primeiro valor a ser inserido é o primeiro a ser removido

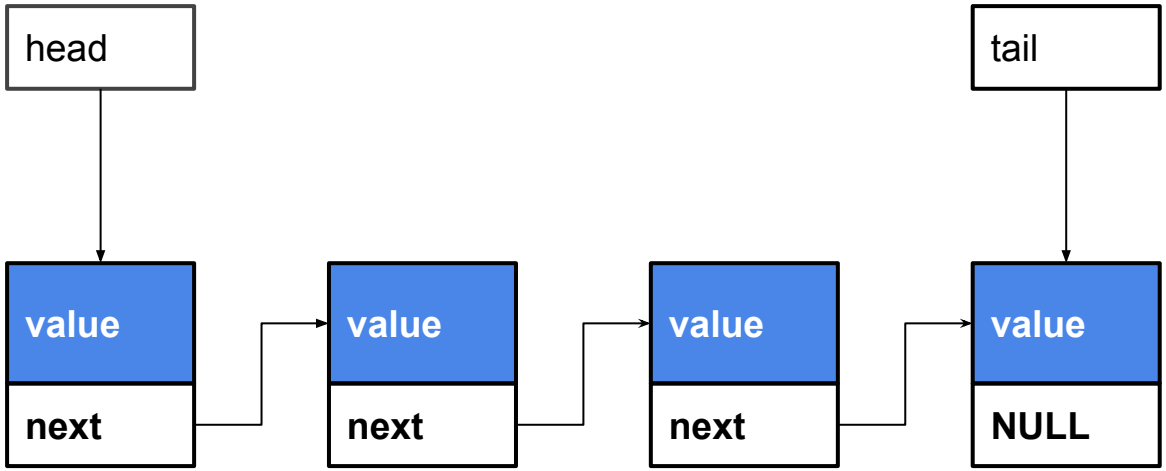
- Fila do Banco
- Fila da Impressora
- Fila do Cinema

TAD

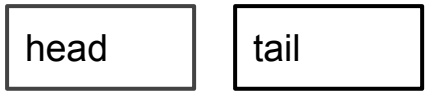
- Bastante similar a ideia de listas da aula passada
- Não vamos nos preocupar com inserir e remover do meio
- Novamente, pode ser implementado com vetores
- Vamos fazer com apontadores!

```c
#ifndef FIFO_H
#define FIFO_H

typedef struct fifo_node {
  int value;
  struct fifo_node *next;
} fifo_node_t;

typedef struct {
  fifo_node_t *head;
  fifo_node_t *tail;
} fifo_t;

fifo_t *fifoCreate();
void fifoInsert(fifo_t *fifo, int value);
int fifoRemove(fifo_t *fifo);
int fifoIsEmpty(fifo_t *fifo);
void fifoFree(fifo_t *fifo);

#endif
```

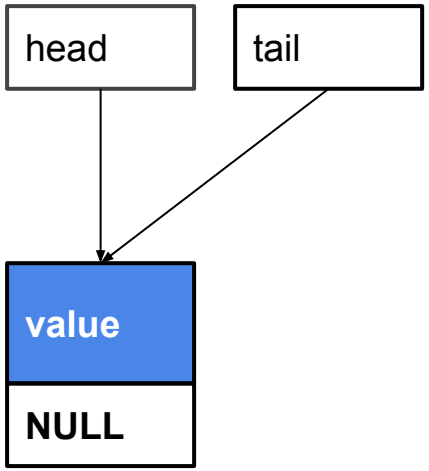
Representando a FIFO



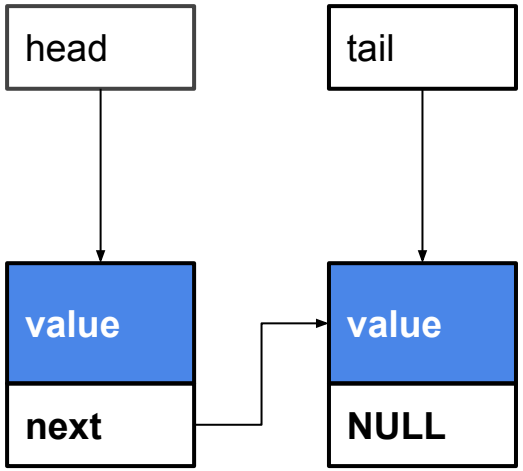
Representando a FIFO



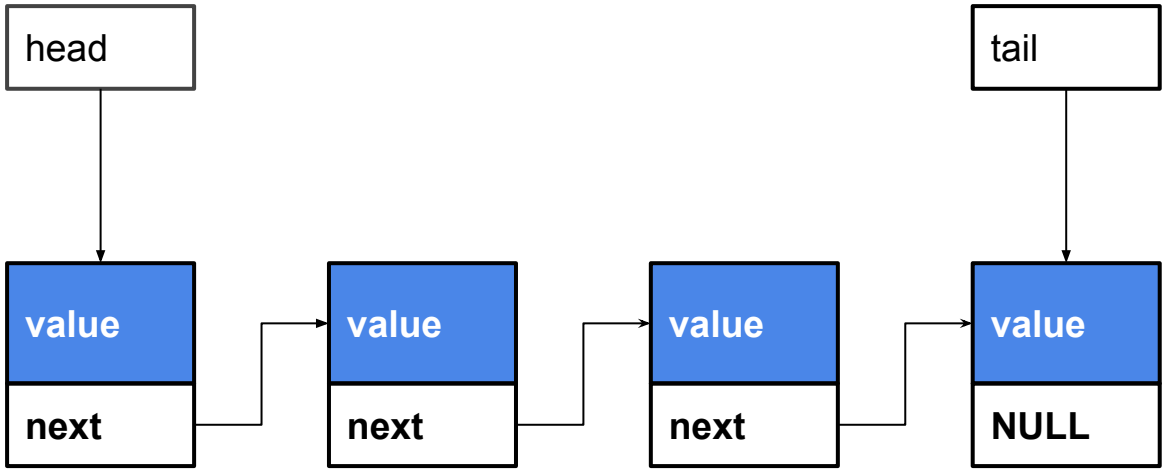
Representando a FIFO



Representando a FIFO



Representando a FIFO



fifoCreate

```
fifo_t *fifoCreate() {
  fifo_t *fifo = (fifo_t *) malloc(sizeof(fifo_t));
  if (fifo == NULL) {
     printf("Memory error");
     exit(1);
  }
  fifo->head = NULL;
  fifo->tail = NULL;
  return fifo;
}
```

fifoInsert

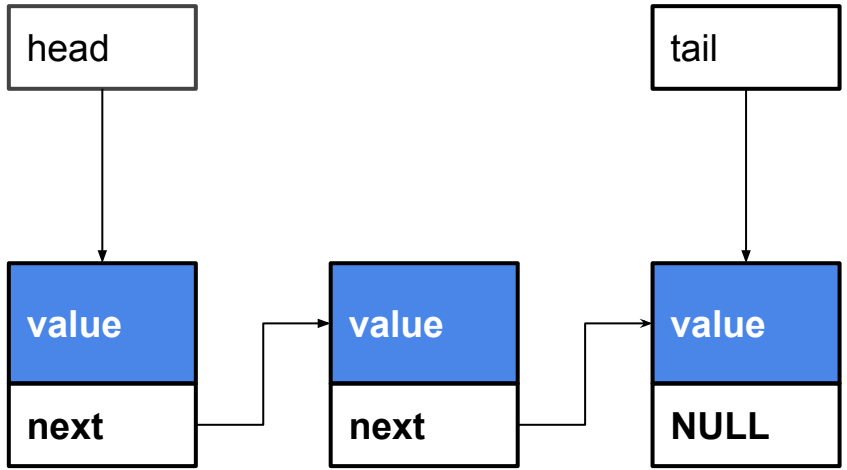
- Alocamos o novo nó
- Atualizamos o fim da fila
- Tratamento para a fila vazia
- O head fica constante utilizamos para remover

fifoInsert

```
void fifoInsert(fifo_t *fifo, int value) {
  fifo_node_t *node = (fifo_node_t *) malloc(sizeof(fifo_node_t));
  if (node == NULL) {
     printf("Memory error");
     exit(1);
  }
  node->value = value;
  node->next = NULL;
  if (fifo->head == NULL) {
     fifo->head = node;
     fifo->tail = node;
  } else {
     fifo->tail->next = node;
     fifo->tail = node;
  }
}
```

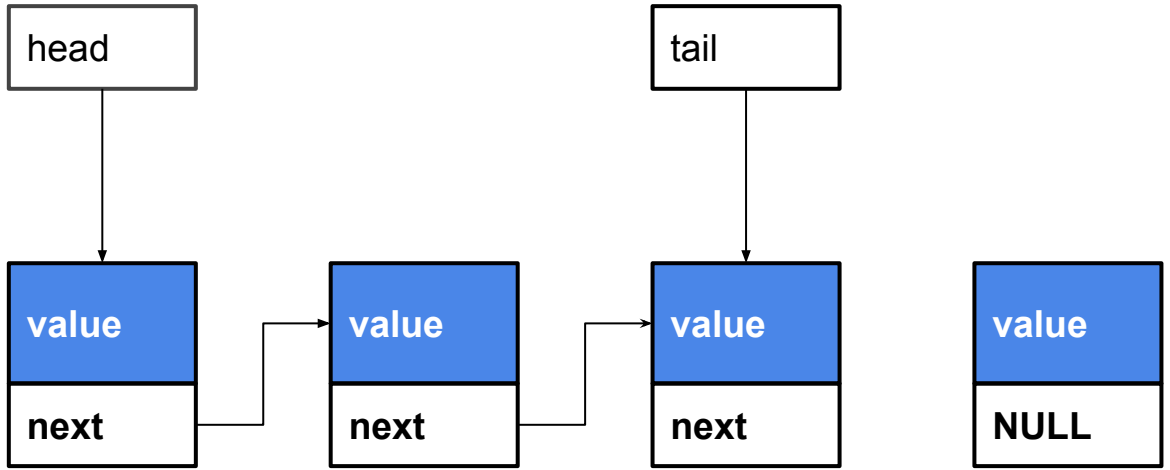
fifoInsert

```
void fifoInsert(fifo_t *fifo, int value) {
  fifo_node_t *node = (fifo_node_t *) malloc(sizeof(fifo_node_t));
  if (node == NULL) {
     printf("Memory error");
     exit(1);
  }
  node->value = value;
  node->next = NULL;
  if (fifo->head == NULL) {
     fifo->head = node;
     fifo->tail = node;
  } else {
     fifo->tail->next = node;
     fifo->tail = node;
  }
}
```



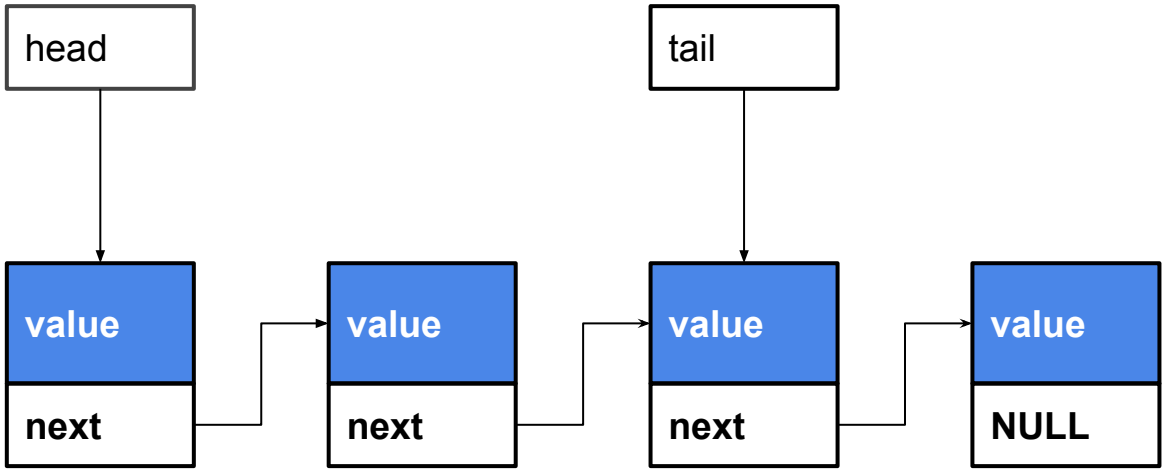
fifoInsert

```
void fifoInsert(fifo_t *fifo, int value) {
  fifo_node_t *node = (fifo_node_t *) malloc(sizeof(fifo_node_t));
  if (node == NULL) {
     printf("Memory error");
     exit(1);
  }
  node->value = value;
  node->next = NULL;
  if (fifo->head == NULL) {
     fifo->head = node;
     fifo->tail = node;
  } else {
     fifo->tail->next = node;
     fifo->tail = node;
  }
}
```



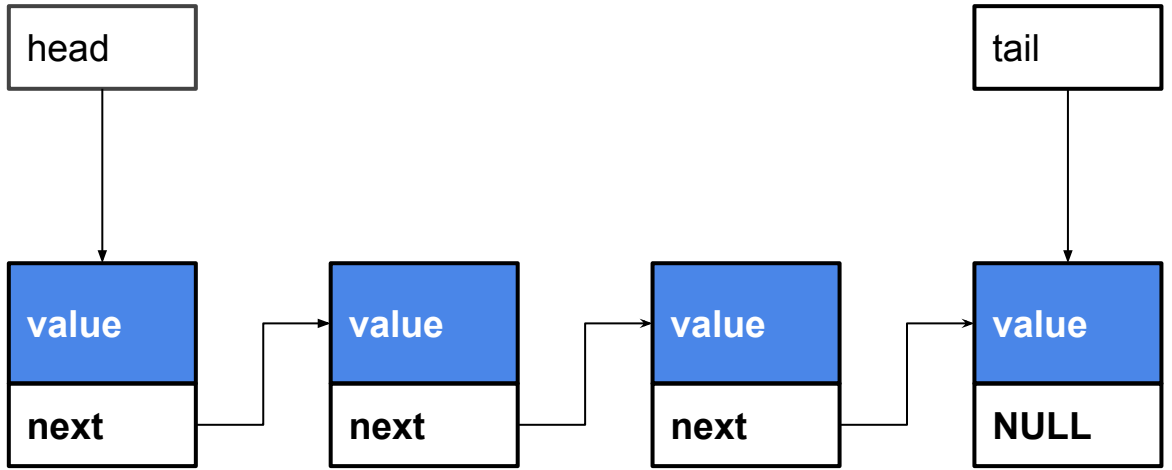
fifoInsert

```
void fifoInsert(fifo_t *fifo, int value) {
  fifo_node_t *node = (fifo_node_t *) malloc(sizeof(fifo_node_t));
  if (node == NULL) {
     printf("Memory error");
     exit(1);
  }
  node->value = value;
  node->next = NULL;
  if (fifo->head == NULL) {
     fifo->head = node;
     fifo->tail = node;
  } else {
     fifo->tail->next = node;
     fifo->tail = node;
  }
}
```



fifoInsert

```
void fifoInsert(fifo_t *fifo, int value) {
  fifo_node_t *node = (fifo_node_t *) malloc(sizeof(fifo_node_t));
  if (node == NULL) {
     printf("Memory error");
     exit(1);
  }
  node->value = value;
  node->next = NULL;
  if (fifo->head == NULL) {
     fifo->head = node;
     fifo->tail = node;
  } else {
     fifo->tail->next = node;
     fifo->tail = node;
  }
}
```



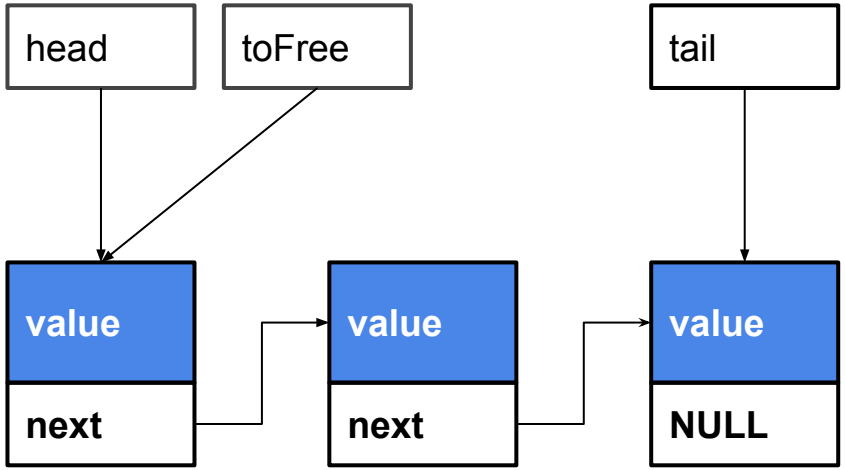
fifoRemove

- Removemos o primeiro nó
- Atualizamos o head
- Tratamento para a fila vazia
- free
- Retornamos o valor

```
int fifoRemove(fifo_t *fifo) {
  fifo_node_t *toFree;
  int toReturn;
  if (fifo->head != NULL) {
     toReturn = fifo->head->value;
     toFree = fifo->head;
     fifo->head = fifo->head->next;
     free(toFree);
  } else {
     printf("FIFO is empty");
     exit(1);
  }
  return toReturn;
}
```

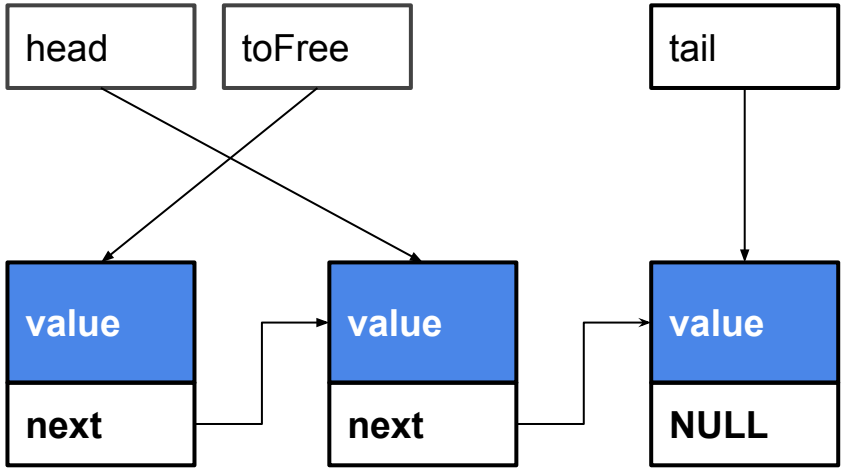
fifoRemove

```
int fifoRemove(fifo_t *fifo) {
  fifo_node_t *toFree;
  int toReturn;
  if (fifo->head != NULL) {
     toReturn = fifo->head->value;
     toFree = fifo->head;
     fifo->head = fifo->head->next;
     free(toFree);
  } else {
     printf("FIFO is empty");
     exit(1);
  }
  return toReturn;
}
```



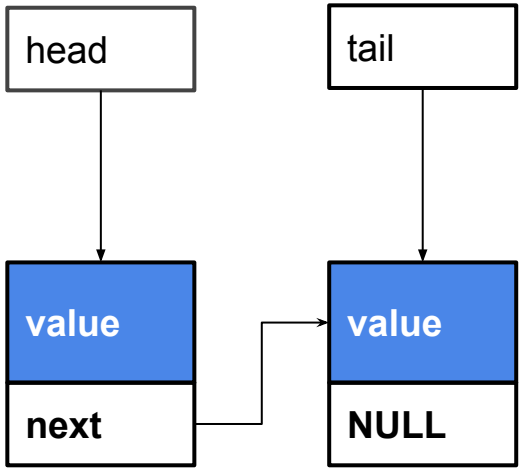
fifoRemove

```
int fifoRemove(fifo_t *fifo) {
  fifo_node_t *toFree;
  int toReturn;
  if (fifo->head != NULL) {
     toReturn = fifo->head->value;
     toFree = fifo->head;
     fifo->head = fifo->head->next;
     free(toFree);
  } else {
     printf("FIFO is empty");
     exit(1);
  }
  return toReturn;
}
```



fifoRemove

```
int fifoRemove(fifo_t *fifo) {
  fifo_node_t *toFree;
  int toReturn;
  if (fifo->head != NULL) {
     toReturn = fifo->head->value;
     toFree = fifo->head;
     fifo->head = fifo->head->next;
     free(toFree);
  } else {
     printf("FIFO is empty");
     exit(1);
  }
  return toReturn;
}
```



Funções Restantes

- fifoIsEmpty
  - Similar ao da pilha
- fifoFree
  - Similar ao da pilha

- Ver código em: https://github.com/flaviovdf/AEDS2-2017-1/tree/master/exemplos/filaspilhas

Custos

- fifoCreate
  - O(1)
- fifoInsert
  - O(1)
- fifoRemove
  - O(1)
- fifoFree
  - O(n)
- fifoIsEmpty
  - O(1)