Notas de aula - 18/04

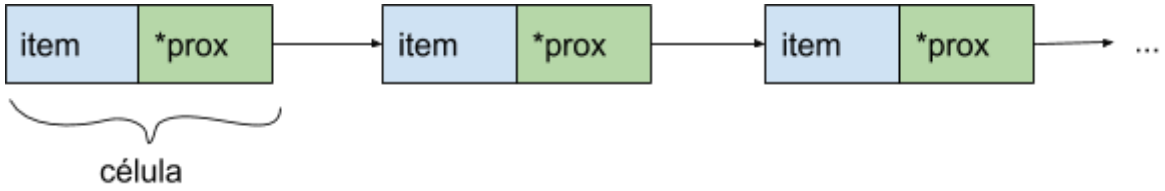
LISTA ENCADEADA

Na lista implementada usando arranjos/vetores, os itens da lista ficavam em posições contíguas de memória. Além disso, precisávamos saber previamente qual seria o tamanho máximo da lista, para alocar a memória necessária no momento da criação da lista.

Esse tipo de implementação possuía alguns inconvenientes:

- Era necessário saber previamente o tamanho máximo da lista. Se o tamanho máximo fosse muito maior do que o utilizado, sobraria memória. Se o tamanho máximo fosse menor do que o utilizado, faltaria memória.
- Para adicionar ou remover itens do meio da lista, era preciso deslocar todos os itens após o item adicionado/removido.

Uma solução a estes problemas, é implementar uma lista encadeada (*linked list*). Uma lista encadeada é uma sequência de células: cada célula contém um item e um ponteiro com o endereço da próxima célula.



Já que cada célula é um ponteiro, cada célula pode ser alocada dinamicamente conforme a necessidade, ou seja: não é mais necessário definir um tamanho máximo para a lista e alocar uma quantidade fixa de memória. Isto é muito vantajoso quando não sabemos previamente quantos itens a lista terá, pois os itens poderão ser adicionados conforme a demanda.

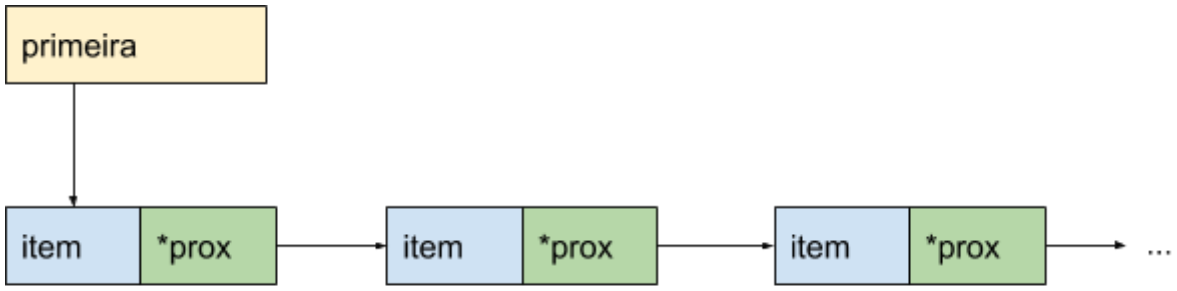
Perceba também que como a alocação de memória é feita para cada célula separadamente, elas não ocupam mais posições contíguas de memória, diferente da lista vista anteriormente.

Estrutura da lista encadeada

```
typedef struct item Item;
typedef struct celula Celula;
typedef struct lista Lista;

// item que vai ser guardado na lista (igual na lista por arranjo)
struct item {
  int chave;
  // demais campos;
};

// a célula contém um item e um ponteiro para a próxima célula da lista
struct celula {
```

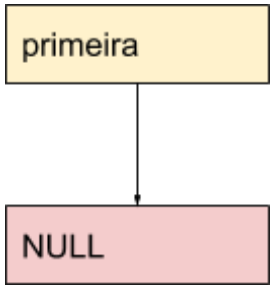
```
  Item item;
  Celula *prox;
};

/* a struct lista guarda o endereço da primeira célula de lista. A
partir da primeira é possível acessar todas as outras */
struct lista {
  Celula *primeira;
};
```

A estrutura lista contém nada mais do que um ponteiro que terá o endereço da primeira célula da lista (algumas implementações mantém o endereço da primeira e da última também). Podemos então visualizar a lista da seguinte forma:



Criação de lista vazia

Se a lista está vazia, não temos um endereço para o qual "primeira" deve apontar. Assim, podemos fazer "primeira" receber um ponteiro nulo (que não aponta para nada). Em C, temos o NULL.



Pensando nisso, um exemplo de função para criar uma lista vazia seria:

```
Lista * cria_lista_vazia() {
  Lista *l = malloc(sizeof(Lista));
  /* NULL é um ponteiro nulo, pode ser usado para inicializar um
ponteiro
  que ainda não está associado a um endereço de memória */
  l->primeira = NULL;
  return l;
}
```

Verificar lista vazia

Para verificar se a lista está vazia, basta ver se “primeira” ainda aponta para NULL:

```
int verifica_lista_vazia(Lista *l) {
  // se a primeira ainda aponta para NULL, então está vazia
  return l->primeira == NULL;
}
```

Inserção na lista

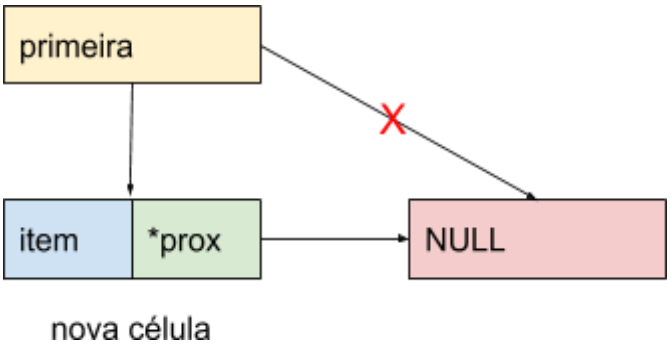
Para adicionar um novo item na lista encadeada, não precisamos nos preocupar se a lista está cheia ou não, em listas encadeadas, o conceito de lista cheia não existe, já que não precisamos definir um tamanho máximo para a lista. Os itens vão sendo adicionados um a um conforme a necessidade. O tamanho máximo da lista vai ser ditado pela memória disponível.

A inserção de itens em uma lista pode ser feita:

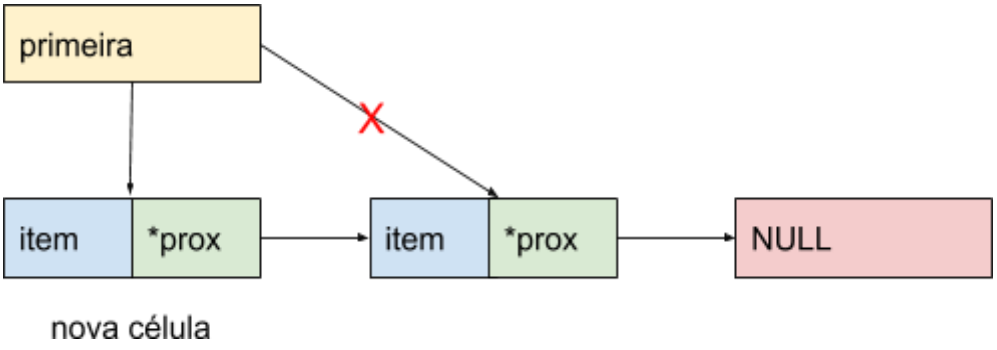
- no início;
- no meio (posição qualquer);
- no fim.

Inserção no início da lista

Considere que vamos inserir um item na lista que estava vazia. Precisamos criar a nova célula, fazer a nova célula apontar aquela que era a primeira (por enquanto é NULL) e então fazer com que “primeira” aponte para nova célula. Vamos manter o ponteiro NULL, pois ele será útil para indicar o término da lista.



E assim sucessivamente:

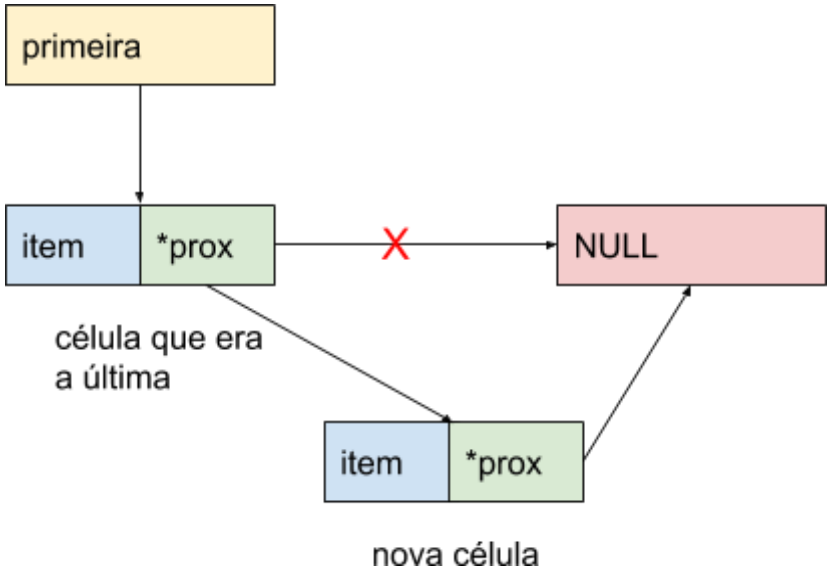


Em C, um exemplo de inserção no início da lista seria:

```
void insere_inicio_lista(Lista *l, int chave) {
  // cria novo item que vai ser guardado na lista
  Item novo;
  novo.chave = chave;
  // cria nova célula que vai guardar o item
  Celula *nova = malloc(sizeof(Celula));
  nova->item = novo;
  // inserção no início - a próxima célula é aquela que era a 1a
  nova->prox = l->primeira;
  // a primeira agora é a nova célula
  l->primeira = nova;
}
```

Inserção no final da lista

Neste caso a nova célula deve ser inserida após a última, que agora vai apontar para a nova célula, que aponta para NULL:

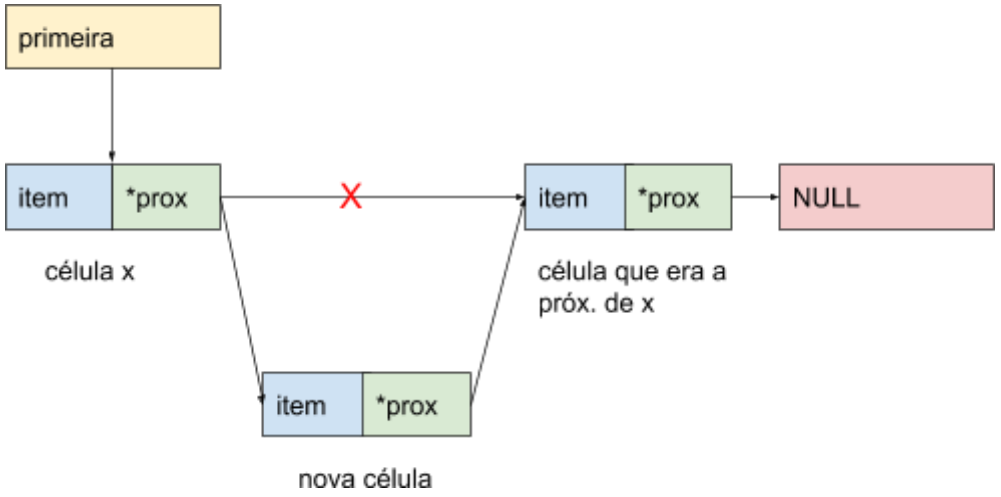



```
void insere_fim_lista(Lista *l, int chave) {
  // cria novo item que vai ser guardado na lista
  Item novo;
  novo.chave = chave;
  // cria nova célula que vai guardar o item
  Celula *nova = malloc(sizeof(Celula));
  nova->item = novo;
  // a nova célula vai ser a última, então após ela tem NULL
  nova->prox = NULL;
  /* variável auxiliar que vai guardar o endereço da última célula
  para inserir a nova depois da última */
  Celula *ultima;
```

```
  // partindo da primeira célula, percorrer a lista até achar a última
  ultima = l->primeira;
  while(ultima->prox != NULL) {
      ultima = ultima->prox;
  }
  // após a última, inserir a nova célula
  ultima->prox = nova;
}
```

Inserção no meio da lista

Como as células da lista não ficam em posições contíguas, acesso por meio de índices não existe. Se quisermos inserir uma nova célula após uma célula x, será preciso encontrar a célula x, ligar a célula x à nova célula, e ligar a nova célula àquela que estava ligada à x.



O mais comum é fazer uma inserção após um item que contém determinada chave. Assim, convém criar uma função que busca a célula que contém o item com essa chave, para que o novo item seja inserido após essa célula:

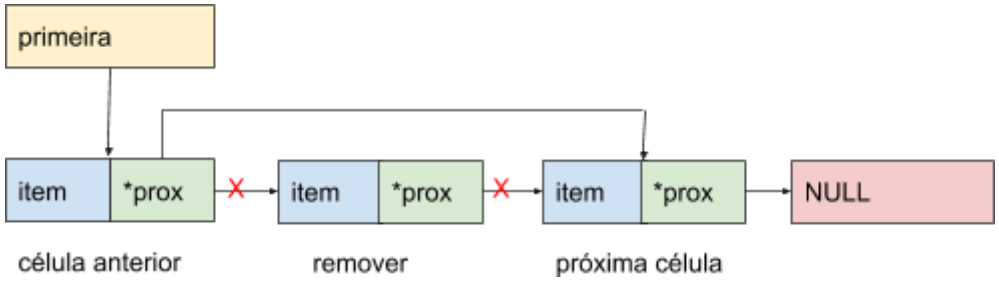
```
/* retorna a célula contendo o item correspondente a chave informada
ou NULL se não encontrou */
Celula * busca_por_chave(Lista *l, int chave) {
  int achou = 0;
  Celula *aux = l->primeira;
  while(achou == 0 && aux != NULL) {
      if(aux->item.chave == chave)
         achou = 1;
      else
         aux = aux->prox;
  }
  return aux;
}
```

Agora é possível usar essa função para fazer a inserção:

```c
void insere_meio_lista(Lista *l, int chave, int x) {
  // cria novo item que vai ser guardado na lista
  Item novo;
  novo.chave = chave;
  // cria nova célula que vai guardar o item
  Celula *nova = malloc(sizeof(Celula));
  nova->item = novo;
  // acha a célula após a qual será feita a inserção
  Celula *aux = busca_por_chave(l, x);
  if(aux != NULL) {
      /* a nova célula vai ser inserida depois de aux, então sua próxima
célula será aquela que era a próxima de aux */
      nova->prox = aux->prox;
      /* a nova célula vai ser inserida depois de aux, então a próxima
célula de aux é a nova célula */
      aux->prox = nova;
  }
  else {
      printf("O item informado não existe.\n");
  }
}
```

Remoção de item

A remoção de um item também pode ser feita do início, meio ou fim. Vamos considerar a remoção do meio que é a mais trabalhosa. Vamos usar a mesma lógica da inserção no meio: buscar a célula contendo um item com certa chave, e remover a célula que vem depois. A célula anterior à célula a ser removida precisa agora ser ligada àquela que é a célula seguinte a célula a ser removida. Finalmente, removemos a célula (liberamos a memória que ela ocupava).



Um exemplo de implementação dessa função:

```c
// remove item após determinada posição
void remove_item(Lista *l, int x) {
  int tamanho = tamanho_lista(l);
  Celula *anterior = busca_por_chave(l, x);
  if(verifica_lista_vazia(l) || anterior == NULL) {
      printf("Erro: a lista está vazia ou o item não existe.\n");
```

```
        return;
    }
    // será removida a célula após aquela contendo a chave buscada
    Celula *remover = anterior->prox;
    // encontra a próxima
    Celula *proxima = remover->prox;
    // liga a anterior àquela que era a próxima da que vai ser removida
    anterior->prox = proxima;
    // libera memória
    free(remover);
}
```

Libera lista

Por fim precisamos liberar a lista. Novamente vamos percorrer toda a lista e ir liberando célula por célula e então a estrutura que aponta para a primeira célula:

```
void libera_lista(Lista *l) {
    Celula *aux = l->primeira;
    Celula *liberar;
    while(aux != NULL) {
        liberar = aux;
        aux = aux->prox;
        free(liberar);
    }
    free(l);
```

Outras funções úteis

A função imprime lista é fundamental para acompanhar o estado da lista. Para imprimir a lista é necessário percorrer toda a lista partindo da primeira célula:

```
void imprime(Lista *l) {
        Celula *aux;
        // partindo da primeira célula, percorrer a lista até achar a
última
        for(aux = l->primeira; aux != NULL; aux = aux->prox)
        printf("chave = %d\n", aux->item.chave);
}
```

Caso seja necessário saber o tamanho da lista, podemos fazer uma função semelhante que incrementa um contador:

```
int tamanho_lista(Lista *l) {
    Celula *aux = l->primeira;
    int i = 0;
    while(aux != NULL) {
        aux = aux->prox;
```

```
        i++;
    }
    return i;
}
```

A implementação das funções acima encontra-se no github.

Tarefa

Como você faria para:

- Remover do início;
- Remover do fim;

Vantagens da lista encadeada

- Não é preciso definir tamanho máximo, as células são alocadas conforme a demanda.
- É possível liberar uma única célula, não é necessário esperar para liberar a lista toda, como na implementação por arranjos/vetores;
- Para remover ou adicionar uma célula, não é preciso deslocar as outras.

Desvantagens da lista encadeada

- Não existe acesso por meio de índices, para encontrar uma célula é preciso percorrer a lista.
- Usa mais memória, pois precisa guardar o item e um ponteiro para a próxima célula.