```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr

    // process user action
    c = getchar();   // get new action
    if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
    if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
    if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
    if (c == commands[3]) moves_to_go=0;

    // scroll down
    if (!moves_to_go--)
      {
        if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
          {
            cramp_tetromino();
            remove_complete_lines();
            cur = &tetrominoes[rand() % NUM_POSES];
            state[0] = (WIDTH - cur->width)/2;
```


Listas Encadeadas

Revisão: Alocação estática de memória

```
int n; char c, int v[30], char str[100];
```

Aloca bloco de memória no espaço de pilha do programa; tamanho a ser alocado é determinado por tipo; desalocação é gerenciada pelo compilador após saída de escopo.

Mais eficiente e segura que alocação dinâmica, mas não permite:

```c
int n;
scanf("Quantas letras? %d", &n);
/* erro de compilação */
char s[n+1];
printf("Palavra: ");
fgets(s, n, stdin);
```

```c
int* cria_vetor(int x, int n) {
   int i, v[100];
   for (i=0; i < n; i++)
      v[i] = x;
   /* devolve ponteiro para bloco
      alocado */
   return v;
}
int main() {
 int *v;
 /* valor de v é indefinido */
 v = cria_vetor(5,10);
 return 0;
}
```

Revisão: Alocação dinâmica de memória

```
void *malloc (unsigned int n);
```

Aloca um bloco de `n` *bytes* consecutivos e retorna um ponteiro para o espaço alocado ou `NULL` em caso de erro

```
void free (void *ptr);
```

Libera o espaço de memória endereçado por `ptr`

Definidos na biblioteca `stdlib.h`

```
int n;
scanf("Quantas letras? %d", &n);
char* s = malloc(n+1);
printf("Palavra: ");
fgets(s, n, stdin);
/* ... */
free(s);
```

```
int* cria_vetor(int x, int n) {
  int* v = malloc(n*sizeof(int));
  for (int i=0; i < n; i++)
     v[i] = x;
  return v;
}
int main() {
 int *v = cria_vetor(5,10);
 /* ... */
 free(v)
 return 0;
}
```

Revisão: Alocação estática de registros

```c
typedef struct { int num, den; } racional;

racional a, b; /* aloca registros */
/* Define a = 1/3 */
a.num = 1;
a.den = 3;

racional *p = &b; /* ponteiro para registro b */
/* Define b = 2/5 */
p->num = 2;
p->den = 5;
```

Revisão: Alocação dinâmica de registros

```c
typedef struct { int num, den; } racional;

racional *cria_racional(int n, int d) {
 racional *q; /* ponteiro para racional */
 q = malloc (sizeof (racional)); /* aloca 1 racional */
            /* com valores de num e den indefinidos */
 q->num = n; q->den = d;  /* define q = n/d */
 return q;
}

int main() {
 /* cria racional r=1/3 */
 racional *r = cria_racional(1,3);
 /* ... */
 free (r);  /* libera memória */
 return 0;
}
```

Listas encadeadas: Motivação

Suponha que estamos projetando um editor de texto usando uma string `txt` para representar o texto sendo editado

Suponha que queremos alterar o texto de

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
txt	M	A	C	1	2	2	␣	é	␣	l	e	g	a	l	\0	?	?	?	?

para

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
txt	M	A	C	1	2	2	␣	n	ã	o	␣	é	␣	l	e	g	a	l	\0

Precisamos deslocar para a direita a parte após é e inserir `não␣` no lugar correto (gastando tempo proporcional ao tamanho do texto)

De forma análoga, suponha que queremos alterar o texto de

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
txt	M	A	C	1	2	2		n	ã	o		é		l	e	g	a	l	\0

para

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
txt	M	A	C	1	2	2		é		l	e	g	a	l	\0	g	a	l	\0

Precisamos agora deslocar para a esquerda a parte após é
(gastando novamento tempo proporcional ao tamanho do texto)

Listas encadeadas: Motivação

Lembrando que inserção e remoção no final de vetor é eficiente, podemos, em vez de percorrer a string sequencialmente, usar um vetor de sequências próximas posições `prox` para representar o texto com edições.

Exemplo:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
prox	1	2	3	4	5	6	7	8	9	10	11	12	13	14	14	16	17	18	19

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
txt	M	A	C	1	2	2		é		l	e	g	a	l	\0	?	?	?	?

⇓ inserção de `não␣` ⇓

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
prox	1	2	3	4	5	6	15	8	9	10	11	12	13	14	14	16	17	18	7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
txt	M	A	C	1	2	2		é		l	e	g	a	l	\0	n	ã	o	

Listas encadeadas: Motivação

Lembrando que inserção e remoção no final de vetor é eficiente, podemos, em vez de percorrer a string sequencialmente, usar um vetor de sequências próximas posições `prox` para representar o texto com edições.

Exemplo:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
prox	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	18

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
txt	M	A	C	1	2	2		n	ã	o		é		l	e	g	a	l	\0

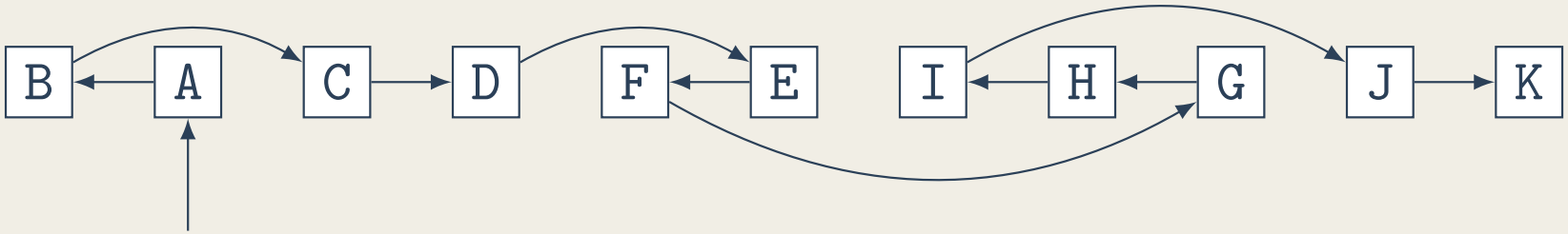
⇓ remoção de `não` ⇓

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
prox	1	2	3	4	5	6	11	8	9	10	11	12	13	14	15	16	17	18	18

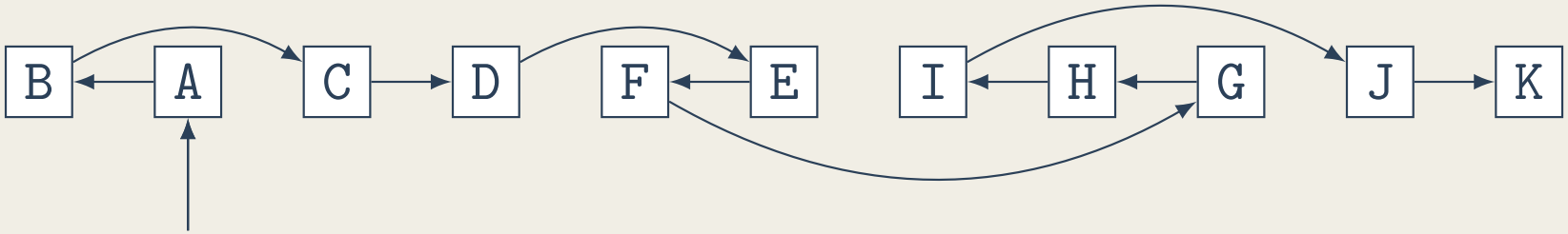
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
txt	M	A	C	1	2	2		n	ã	o		é		l	e	g	a	l	\0

Agora suponha que você queira a ordenar os caracteres na sentença em ordem alfabética abaixo sem reescrever a frase, indicando a próxima letra a ser lida:

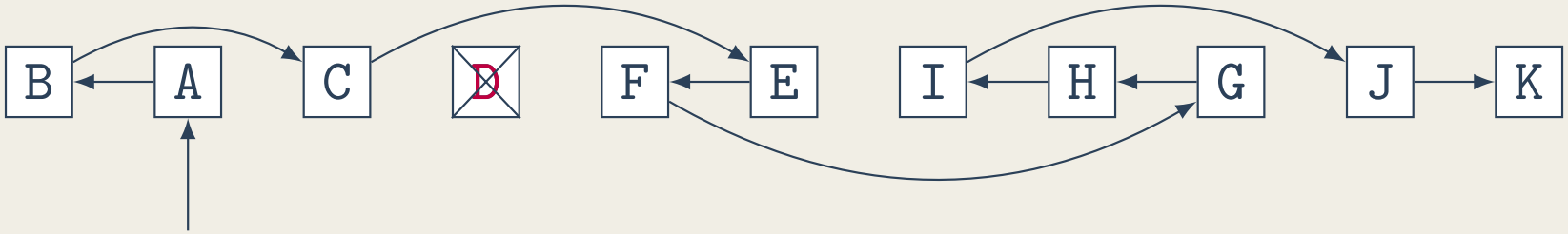
Agora suponha que você queira a ordenar os caracteres na sentença em ordem alfabética abaixo sem reescrever a frase, indicando a próxima letra a ser lida:



Agora suponha que você queira remover D sem alterar a ordem:

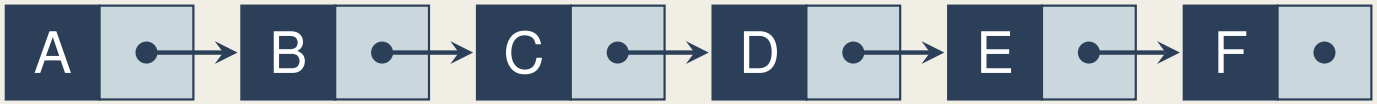


Agora suponha que você queira remover D sem alterar a ordem:



Listas encadeadas: Definição

Estrutura de dados *dinâmicas* e *autoreferenciais* usadas para representar sequências eficientemente editáveis



- Elementos são organizados em células
- Flechas são representados por ponteiros

```c
typedef struct celula_st {
    char valor;                 /* conteúdo */
    struct celula_st *prox;     /* para próxima célula */
} celula; /* tipo de dado */
```

Primeira célula armazena conteúdo do primeiro elemento


```
1 celula *c1 = malloc(sizeof(celula));
2 celula *c2 = malloc(sizeof(celula));
3 celula *c3 = malloc(sizeof(celula));
4 c1->valor = 'A'; c1->prox = c2;
5 c2->valor = 'B'; c2->prox = c3;
6 c3->valor = 'C'; c3->prox = NULL;
```

Lista vazia é representada por ponteiro nulo: `c1 = NULL`

Segunda célula armazena conteúdo do primeiro elemento


```
1 celula *cabeca = malloc(sizeof(celula));
2 celula *c1 = malloc(sizeof(celula));
3 celula *c2 = malloc(sizeof(celula));
4 celula *c3 = malloc(sizeof(celula));
5 cabeca->prox = c1;
6 c1->valor = 'A'; c1->prox = c2;
7 c2->valor = 'B'; c2->prox = c3;
8 c3->valor = 'C'; c3->prox = NULL;
```

Lista vazia é representada por cabeça apontando para nulo:

`cabeca->prox = NULL`

Lista encadeada: Iteração

```c
/* Imprime lista sem cabeça */
void Imprime (celula *lista) {
  celula *p = lista;
  while (p != NULL) {
      printf("%c ", p->valor);
      p = p->prox;
   }
}
```

```c
/* Imprime lista com cabeça */
void Imprime (celula *lista) {
  celula *p = lista->prox;
  while (p != NULL) {
      printf("%c ", p->valor);
      p = p->prox;
   }
}
```

Alternativas:

```c
/* Imprime lista sem cabeça */
void Imprime (celula *lista) {
  for (celula *p = lista; p != NULL; p = p->prox)
      printf("%c ", p->valor);
}
```

```c
/* Imprime lista com cabeça */
void Imprime (celula *lista) {
  for (celula *p = lista->prox;
       p != NULL;
       p = p->prox)
      printf("%c ", p->valor);
}
```

Lista encadeada: Busca linear

```
1 /* Recebe um caractere x e uma lista com cabeça
2    Devolve o endereço de um célula que contém x
3    ou NULL em caso de falha */
4 celula *Busca (char x, celula *lista) {
5   celula *p = lista->prox;
6   while (p != NULL && p->valor != x)
7      p = p-> prox;
8   return p;
9 }
```

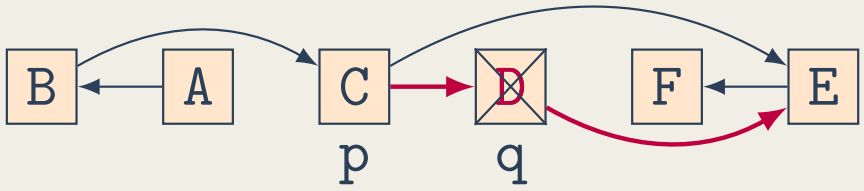
Implementação alternativa:

```
1 /* Recebe um caractere x e uma lista com cabeça
2    Devolve o endereço de um célula que contém x
3    ou NULL em caso de falha */
4 celula *Busca (char x, celula *lista) {
5   for (celula *p = lista->prox;
6         p != NULL && p->valor != x;
7         p = p->prox);
8   return p;
9 }
```

Lista encadeada: Remoção

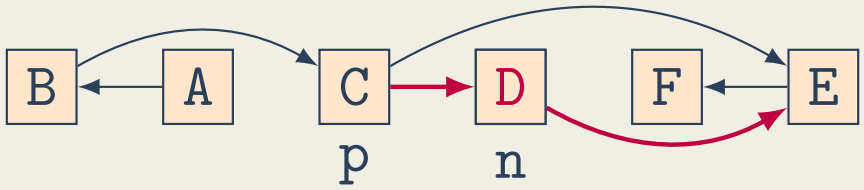
- Redirecione flecha de entrada para flecha de saída
- Libere memória alocada

```
1 /* Recebe o endereço p de um célula e remove p->prox
2    Assume que p != NULL e p->prox != NULL */
3 void Remove (celula *p) {
4   celula *q;
5   q = p->prox;
6   p->prox = q->prox;
7   free (q);
8 }
```



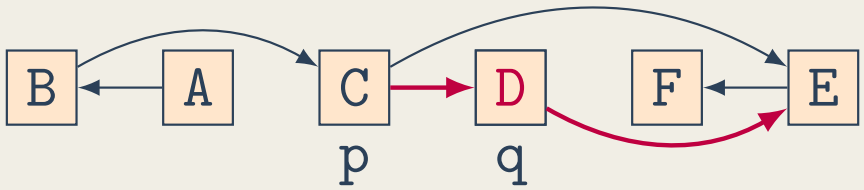
Lista encadeada: Inserção

```c
/* Cria e insere uma nova célula de valor x
   logo após uma dada célula p
   (assume que p != NULL) */
celula *Insere (char x, celula *p) {
  celula *n = malloc (sizeof (celula));
  n->valor = x;
  n->prox = p->prox;
  p->prox = n;
  return n;
}
```



Lista encadeada: Busca seguida de remoção

```
1  /* Remove de uma lista com cabeça
2     a primeira célula cujo valor é x,
3     se existir */
4  celula *BuscaERemove (char x, celula *lista) {
5    celula *p, *q;
6    p = lista; q = lista->prox;
7    while (q != NULL && q->valor != x) {
8        p = q; q = q->prox;
9    }
10   if (q != NULL) {
11      p->prox = q->prox;
12      free (q);
13   }
14   return p;
15 }
```



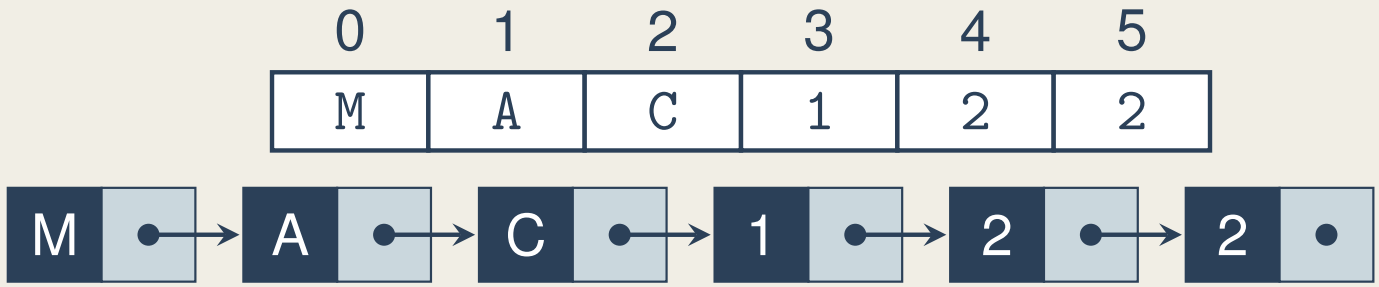
QUESTÃO

Como concatenar duas listas com cabeça?

Como concatenar duas listas com cabeça?


```c
/* Concatena listas e devolve cabeça da lista resultante
   Modifica lista1 */
celula *concatena (celula *lista1, celula *lista2) {
  celula *p = lista1;
  while (p->prox != NULL)
      p = p->prox;
  p->prox = lista2->prox;
  return lista1;
}
```

CONVERTENDO STRING EM LISTA ENCADEADA




```c
/* String para lista sem cabeça */
celula *string_para_lista (char s[]) {
  celula *c = NULL, *p;
  if (s[0] != '\0') { /* se string não for vazia*/
    c = malloc(sizeof(celula)); /* cria cabeça  */
    c->valor = s[0];   /* e guarda para retorno */
    s++;   p = c; /* ptr para cauda da lista */
    while (*s != '\0') {
      p->prox = malloc(sizeof(celula));
      p = p->prox;      p->valor = s;     s++;
    }
  }  p->prox = NULL; /* fim da lista */
  return c; /* devolve ptr para cabeça da lista */
}
```

VETOR

- Espaço proporcional a número de elementos e tipo de conteúdo
- Inserção/Remoção no final em tempo constante
- Inserção/Remoção no começo requer `n` deslocamentos
- Acesso aleatório em tempo constante

LISTA ENCADEADA

- Espaço proporcional a número de elementos, tipo de conteúdo e número de flechas (ponteiros)
- Inserção/Remoção em qualquer lugar em tempo constante (se tivermos ponteiro para célula respectiva)
- Acesso aleatório em tempo linear
- Fácil extensão para conteúdo complexo (strings, racionais, imagens etc)

Para casa

Exercícios: 7A, 7B e 7C