Estruturas de dados dinâmicas

- Lista
- Árvore binária
- Vetor
- Tabela de dispersão (*hash table*)

Tópicos:

- estruturas auto referenciadas;
- programação genérica baseada em void *;
- aritmética de ponteiros;
- gestão dinâmica da memória;
- funções de biblioteca **malloc** e **free**;
- construção de programas com vários módulos.

A linguagem C não especifica no âmbito da sua biblioteca, contentores genéricos como listas, vetores ou outros. No entanto a linguagem dispõe de mecanismos que permitem a sua criação.

A GLib, desenvolvida pela fundação GNOME, é um exemplo de biblioteca de utilização genérica para a linguagem C. É utilizada como referência na implementação dos contentores genéricos aqui apresentados.

GLib - https://developer.gnome.org/glib/

As fontes dos programas aqui apresentados podem se obtidas em:

https://github.com/econde/psc_code

Listas

São apresentadas cinco implementações de listas ligadas, numa sequência de introdução gradual dos mecanismos e das técnicas de programação em C, por diferença com a implementação anterior.

1. lista simplesmente ligada;
2. lista duplamente ligada com sentinela;
3. lista não intrusiva duplamente ligada;
4. lista não intrusiva duplamente ligada com implementação separada;
5. lista intrusiva duplamente ligada com implementação separada.

Como exemplo de aplicação é realizado um programa de simulação de uma fila de espera, em que a fila é baseada na lista ligada.

1 Lista simplesmente ligada

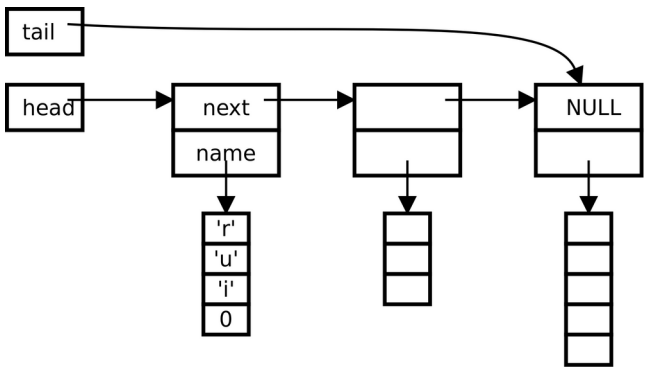
Os nós da lista, implementados pela **`struct User`**, incluem os dados de um utente e a indicação do próximo elemento da lista, que corresponde ao próximo utente na fila.

Os dados do utente são compostos apenas pela indicação do nome, que é armazenado num *array*, na forma de string, alocado dinamicamente. Por isso o campo **`name`** é um ponteiro para *char*. O campo **`next`** é um ponteiro para o próximo elemento na lista, que é do mesmo tipo.

```
typedef struct User {
      struct User *next;
      char *name;
} User;
```

A lista é representada pelo ponteiro **`head`** e **`tail`** que indicam, respetivamente, o primeiro e último elemento da lista. Inicialmente a lista está vazia, por isso os ponteiros são iniciados com **`NULL`**.

```
User *head = NULL, *tail = NULL;
```



Inserir novo utente na fila

Esta operação é realizada em três passos:

1. alocar de memória para uma nova **`struct User`** – linhas 1 a 6;
2. preencher a *struct* com os dados do novo utente – linhas 7 a 13;
3. inserir a *struct* na lista – linhas 15 a 21.

O segundo passo engloba também a alocação do *array* de caracteres para alojar o nome do utente – linha 7.

```
 1 void user_insert(char *name) {
 2       User *user = malloc(sizeof *user);
 3       if (user == NULL) {
 4             fputs(stderr, "Out of memory");
 5             exit(-1);
 6       }
 7       user->name = malloc(strlen(name));
 8       if (user->name == NULL) {
 9             free(user);
10             fputs(stderr, "Out of memory");
11             exit(-1);
12       }
13       strcpy(user->name, name);
14
15       user->next = NULL;
16       if (NULL == head)
17             head = tail = user;
18       else {
19             tail->next = user;
20             tail = user;
21       }
22 }
```

Remover utente da fila por atendimento normal

Esta operação é realizada em três passos:

1. retirar a primeira *struct* da lista – linhas 4 a 6 ;
2. extrair a informação do utente que é apenas o nome – linha 8;
3. libertar a memória alocada para a struct – linha 9.

A memória alocada para alojar o nome não é libertada ainda porque o ponteiro para esse local vai ser retornado pela função.

```
 1 char *user_answer() {
 2       if (NULL == head)
 3             return NULL;
 4       User * user = head;
 5       head = head->next;
 6       tail = NULL == head ? NULL : tail;
 7
 8       char * name = user->name;
 9       free(user);
10       return name;
```

```
11 }
```

Remover utente da fila por desistência

Esta operação é realizada em três passos:

1. localizar na lista a **`struct User`** que representa este utente – linha 4 a 6;
2. retirar essa *struct* da lista – linhas 7 a 14;
3. e por fim libertar a memória alocada – linhas 15 a 17.

A sequência das operações das linhas 15 e 16 terá que ser exatamente esta – não poderia ser invertida. A partir do momento em que se liberta uma porção de memória com a operação **`free`**, deixa de se poder aceder ao conteúdo dessa porção de memória. Se executasse **`free(user)`** em primeiro lugar, ao executar **`free(user->name)`** o conteúdo de **`user->name`** já seria inválido.

```
 1 void user_remove(char *name) {
 2       if (NULL == head)
 3             return;
 4       User *prev = NULL;
 5       for (User *user = head; user != NULL; prev = user, user = user->next)
 6             if (strcmp(name, user->name) == 0){
 7                   if (NULL == prev) {
 8                         head = user->next;
 9                         tail = NULL == head ? NULL : tail;
10                   }
11                   else {
12                         prev->next = user->next;
13                         tail = NULL == prev->next ? prev : tail;
14                   }
15                   free(user->name);
16                   free(user);
17                   return;
18             }
19 }
20
```

Listar utentes presentes na fila

```
1 void user_print() {
2       if (NULL == head) {
3             printf("Fila vazia\n");
4             return;
5       }
6       int i = 1;
7       for (User *user = head; user != NULL; user = user->next)
8             printf("%d: %s\n", i++, user->name);
9 }
```

Vazar fila de utentes

```
1 void user_leak_queue() {
2       for (User *next, *p = head; p != NULL; p = next) {
3             next = p->next;
4             free(p->name);
5             free(p);
```

```
    6       }
    7 }
```

Programa principal

Recolha, interpretação e execução de comandos de utilização do programa.

```
static void help() {
      printf("Comandos:\n"
            "\tS\t\t- Sair\n"
            "\tN <name> \t- Chegada de novo utente\n"
            "\tD <name>\t- Desistencia de utente\n"
            "\tL\t\t- Listar fila de espera\n"
            "\tA\t\t- Atender utente\n");
}

int main() {
      char line[100];
      while (1) {
            if (fgets(line, sizeof(line), stdin) == NULL)
                  exit(1);
            char *command = strtok(line, " \n");
            char *name = strtok(NULL, " \n");
            switch (command[0]) {
                  case 's':
                  case 'S':
                        user_leak_queue();
                        exit(0);
                  case 'h':
                  case 'H':
                        help();
                        break;
                  case 'n':
                  case 'N':
                        user_insert(name);
                        break;
                  case 'd':
                  case 'D':
                        user_remove(name);
                        break;
                  case 'l':
                  case 'L':
                        user_print();
                        break;
                  case 'a':
                  case 'A':
                        name = user_answer();
                        if (NULL == name)
                              printf("Fila vazia\n");
                        else {
                              printf("Atender %s\n", name);
                              free(name);
                        }
                        break;
            }
      }
}
```

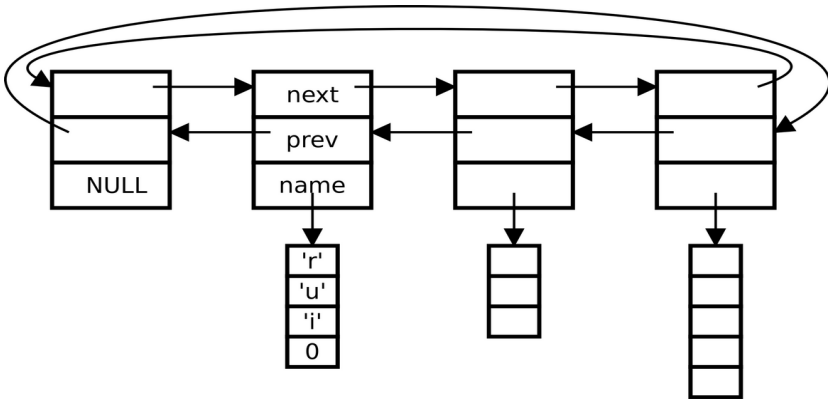
2 Lista duplamente ligada com sentinela

Como no caso anterior, os dados do utente são o nome, armazenado num *array* alocado dinamicamente, apontado pelo campo **`name`**. Os campos **`next`** e **`prev`** são ponteiros para os elementos adjacentes na lista.

```
typedef struct User {
      struct User *next, *prev;
      char *name;
} User;
```

A lista é representada por uma sentinela – **`queue`**, do mesmo tipo dos elementos da lista.
A lista vazia é caraterizada por ter os campos **`next`** e **`prev`** a apontar para o próprio elemento sentinela.

```
User queue = { .next = &queue, .prev = &queue};
```



Inserir um novo utente na fila

Relativamente à versão de lista simplesmente ligada, a diferença está nas operações de manipulação da lista.

Foram removidas as verificações às operações de alocação de memória para facilitar a análise do código.

```
 1 void user_insert(char *name) {
 2       User *user = malloc(sizeof *user);
 3       user->name = malloc(strlen(name));
 4       strcpy(user->name, name);
 5
 6       user->prev = queue.prev;
 7       user->next = &queue;
 8       queue.prev->next = user;
 9       queue.prev = user;
10 }
```

Remover utente da fila por atendimento normal

Numa lista duplamente ligada com sentinela verifica-se se a lista está vazia se o campo **`next`** (ou

prev) apontar para o próprio elemento sentinela.

```
 1 char *user_answer() {
 2       if (queue.next == &queue)
 3             return NULL;
 4       User *user = queue.next;
 5       user->next->prev = user->prev;
 6       user->prev->next = user->next;
 7
 8       char *name = user->name;
 9       free(user);
10       return name;
11 }
```

Remover utente da fila por desistência

```
 1 static void user_remove(char * name) {
 2       if (queue.next == &queue)
 3             return;
 4       for (User *user = queue.next; user != &queue; user = user->next)
 5             if (strcmp(name, user->name) == 0) {
 6                   user->prev->next = user->next;
 7                   user->next->prev = user->prev;
 8                   free(user->name);
 9                   free(user);
10                   return;
11             }
12 }
```

Vazar fila de utentes

```
1 void user_leak_queue() {
2       for (User *next, *p = queue.next; p != &queue; p = next) {
3             next = p->next;
4             free(p->name);
5             free(p);
6       }
7 }
```

3 Lista duplamente ligada genérica não intrusiva

Designa-se por “lista intrusiva” uma lista em que os elementos de formação da lista (campos **next** e **prev**) fazem parte da mesma *struct* que os elementos de dados.

Designa-se por “lista não intrusiva” uma lista cujos os elementos de formação da lista são externos à *struct* que contém os elementos de dados.

Uma *struct* com elementos de dados pode estar inserida simultaneamente em várias listas não intrusivas. Enquanto que em listas intrusivas só pode estar inserida nas listas para as quais tiver elementos de ligação.

Para exemplificar a utilização de lista não intrusiva no programa de simulação de fila de espera, são criadas duas *struct*, uma para os elementos de formação da lista e outra para os elementos de dados.

A formação da lista é baseada na **struct List_node** que contém os campos **next** e **prev** para

referenciar os elementos adjacentes e um ponteiro para a *struct* de dados.

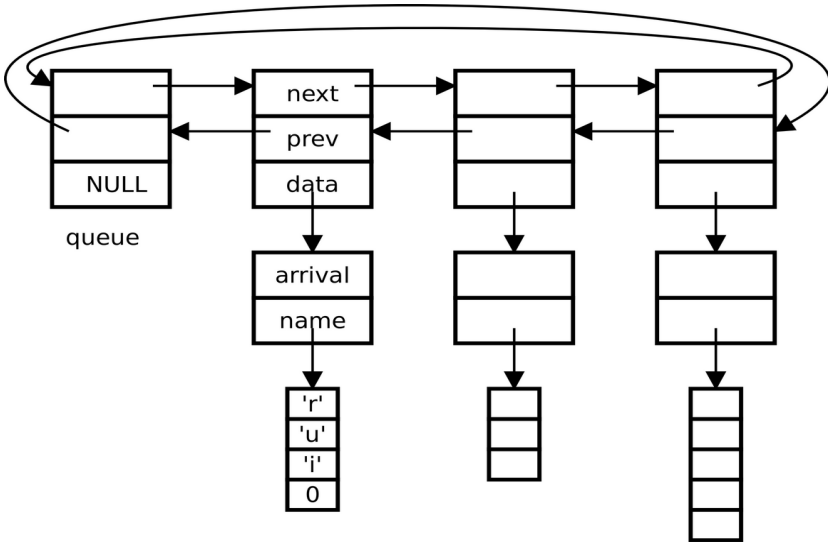
```
typedef struct List_node {
      struct List_node *next, *prev;
      void *data;
} List_node;
```

Os dados do utente são compostos pela indicação do nome e da hora a que chegou à fila. O nome é armazenado num *array* alocado dinamicamente, por isso o campo **`name`** é um ponteiro para *char*. O campo **`arrival`** guarda informação de tempo que é representado pelo número de segundos, desde o início de uma época.

```
typedef struct User {
      time_t arrival;
      char *name;
} User;
```

A lista é representada pela sentinela, do tipo do elemento de formação da lista. A Lista vazia é caraterizada por ter os campos **`next`** e **`prev`** a apontar para o próprio elemento sentinela.

```
static List_node queue = { .next = &queue, .prev = &queue};
```



Inserir um novo utente na fila

Como os elementos de formação da lista são externos, as operações relativas à lista – linhas 7 a 13 – são completamente separáveis das operações de utilizador – linhas 2 a 5.

```
1 void user_insert(char *name) {
2       User *user = malloc(sizeof *user);
3       user->name = malloc(strlen(name));
4       strcpy(user->name, name);
5       time(&user->arrival);
6
7       List_node *node = malloc(sizeof *node);
8       node->data = user;
9
```

```
10        node->next = &queue;
11        node->prev = queue.prev;
12        queue.prev->next = node;
13        queue.prev = node;
14  }
```

Remover utente da fila por atendimento normal

O acesso aos dados do utilizador faz-se através do campo **`data`**. É necessário um *cast* para o tipo **`User`** par se aceder aos dados de utilizador – linha 7.

```
 1  char *user_answer() {
 2        if (&queue == queue.next)
 3              return NULL;
 4        List_node *node = queue.next;
 5        node->next->prev = node->prev;
 6        node->prev->next = node->next;
 7        char *name = ((User *)node->data)->name;
 8        free(node->data);
 9        free(node);
10        return name;
11  }
```

Remover utente da fila por desistência

```
 1  void user_remove(char *name) {
 2        if (&queue == queue.next)
 3              return;
 4        for (List_node *node = queue.next; node != &queue; node = node->next)
 5              if (strcmp(((User *)node->data)->name, name) == 0) {
 6                    node->next->prev = node->prev;
 7                    node->prev->next = node->next;
 8                    free(((User *)node->data)->name);
 9                    free(node->data);
10                    free(node);
11                    return;
12              }
13  }
```

Listar utentes presentes na fila

```
 1  void user_print() {
 2        if (&queue == queue.next) {
 3              printf("Fila vazia\n");
 4              return;
 5        }
 6        for (List_node *node = queue.next; node != &queue; node = node->next)
 7              printf("Nome: %s\n", ((User *)node->data)->name);
 8  }
```

Vazar fila de utentes

```
 1  void user_leak_queue() {
 2       for (List_node *next, *node = queue.next; node != &queue; node = next) {
```

```
3               next = node->next;
4               free(((User *)node->data)->name);
5               free(node->data);
6               free(node);
7       }
8 }
```

4 Lista duplamente ligada genérica não intrusiva em separado

Esta implementação é equivalente à anterior, com a diferenças das operações sobre a lista serem organizadas em funções e separadas noutro ficheiro. O objetivo é viabilizar a reutilização da implementação das operações sobre listas, em outras situações.

A implementação das operações sobre a lista deve ser feita de modo a facilitar a sua utilização, dispensando o utilizador de lidar com detalhes de implementação interna.

O código utilizador da lista apenas tem que invocar a função que realiza a operação desejada e passar os respetivos argumentos.

Tomando como referência a biblioteca GLib[1], é definido o subconjunto de operações estritamente necessárias para o programa de simulação de fila de espera:

- **list_create** – criar uma lista nova
- **list_destroy** – eliminar uma lista
- **list_insert_rear** – inserir elemento no fim da lista
- **list_remove** – eliminar elemento da lista
- **list_data** – obter o campo de dados
- **list_front** – obter o primeiro elemento da lista
- **list_empty** – verificar se a lista está vazia
- **list_search** – procurar por elemento com certas características
- **list_for_each** – realizar ação sobre todos os elementos da lista

O utilizador terá também de conhecer a entidade “nó da lista”, do tipo **`List_node.`**

A lista é representada pelo ponteiro devolvido por **`list_create`** do tipo ponteiro para **`List_node`**.

```
static List_node *queue;
```

Antes de começar a utilizar a lista é necessário invocar a função **`list_create`** e inicializar o ponteiro **`queue`**.

```
queue = list_create();
```

Inserir um novo utente na fila

```
1 void user_insert(char *name) {
2       User *user = malloc(sizeof *user);
3       user->name = malloc(strlen(name));
4       strcpy(user->name, name);
5       time(&user->arrival);
```

1 https://developer.gnome.org/glib/

```
6       list_insert_rear(queue, user);
7 }
```

Remover utente da fila por atendimento normal

```
1 char *user_answer() {
2       if (list_empty(queue))
3             return NULL;
4       List_node *node = list_front(queue);
5       User *user = list_data(node);
6       list_remove(node);
7
8       char *name = user->name;
9       free(user);
10      return name;
11 }
```

Remover utente da fila por desistência

A operação de procura é realizada pela operação genérica **`list_search`** que tem como parâmetro o ponteiro para a função de verificação. Por definição da interface de programação da lista, a função de verificação tem como primeiro parâmetro o ponteiro para o elemento de dados – campo **`data`** de **`List_node`** – e como segundo parâmetro, o segundo parâmetro de **`list_search`**, por onde o utilizador enviará um elemento de contexto à função de verificação – neste caso o nome a procurar.

```
1 int cmp(const void *a, const void *b) {
2       return strcmp(((User *)a)->name, (char *)b);
3 }
4
5 void user_remove(char *name) {
6       if (list_empty(queue))
7             return;
8       List_node *node = list_search(queue, name, cmp);
9       if (node == NULL)
10            return;
11      User *user = list_data(node);
12      list_remove(node);
13      free(user->name);
14      free(user);
15 }
```

Listar utentes na fila

Tal como na procura de utente a operação de listar precisa percorrer os elementos da lista, neste caso, todos. Esta operação é realizada através da operação genérica **`for_each`**, que tem como parâmetro o ponteiro para uma função que invoca em cada iteração. Esta função recebe como parâmetro o ponteiro para o elemento de dados – campo **`data`** de **`List_node.`**

```
1 static void print(void *user) {
2       int n = 1;
3       printf("%d: %s\n", (*(int*)n)++, ((User *)user)->name);
4 }
5
6 void user_print() {
```

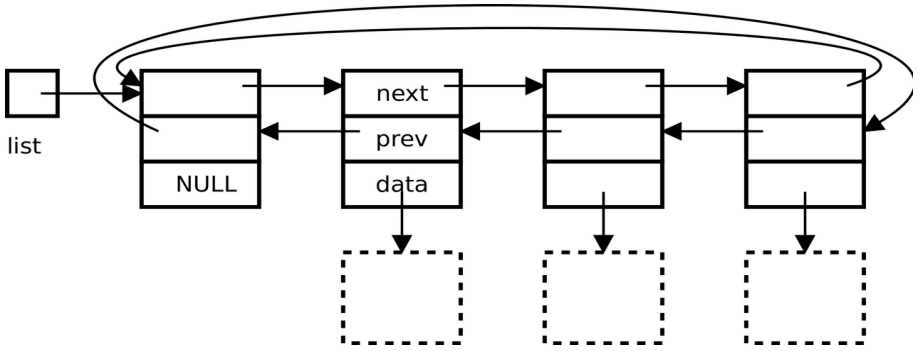
```
 7       if (list_empty(queue)) {
 8             printf("Fila vazia\n");
 9             return;
10       }
11       list_foreach(queue, print);
12 }
```

Vazar fila de utentes

```
 1 void free_user(void *user) {
 2       free(((User *)user)->name);
 3       free(user);
 4 }
 5
 6 void user_leak_queue() {
 7       list_foreach(queue, free_user);
 8       list_destroy(queue);
 9 }
```

Lista genérica, dupla ligação, não intrusiva - implementação

Para se utilizar as operações sobre a lista implementadas na forma de funções em ficheiro separado é preciso conhecer as suas assinaturas. O processo que normalmente se usa na linguagem C consiste em criar dois ficheiros, um apenas com a assinatura das funções – declaração – e outro com a implementação das funções – definição.



listd.h

```
 1 #ifndef LISTD_H
 2 #define LISTD_H
 3
 4 typedef struct List_node {
 5       struct List_node *next, *prev;
 6       void *data;
 7 } List_node;
 8
 9 List_node *list_create();
10
11 void list_destroy(List_node *p);
12
13 int list_insert_rear(List_node *p, void *data);
14
15 void list_remove(List_node *node);
```

```
 16
 17 List_node *list_search(List_node *node, void *data,
 18                        int (*cmp)(const void*, const void*));
 19
 20 void list_foreach(List_node *node, void (*do_it)(void*));
 21
 22 List_node *list_front(List_node *node);
 23
 24 int list_empty(List_node *node);
 25
 26 void * list_data(List_node *node);
 27
 28 #endif
```

listd.h

listd.c

```
  1 #include <stdlib.h>
  2 #include <assert.h>
  3 #include "list_d.h"
  4
  5 void list_init(List_node *node) {
  6       node->next = node->prev = node;
  7 }
  8
  9 List_node *list_create() {
 10       List_node * node = malloc(sizeof(List_node));
 11       if (node == 0)
 12             return 0;
 13       list_init(node);
 14       return node;
 15 }
 16
 17 void list_destroy(List_node *node) {
 18       assert(node);
 19       free(node);
 20 }
 21
 22 int list_insert_rear(List_node *node, void *data) {
 23       List_node *new_node = malloc(sizeof(List_node));
 24       if (new_node == NULL)
 25             return 0;
 26       new_node->data = data;
 27
 28       new_node->prev = node->prev;
 29       new_node->next = node;
 30       node->prev->next = new_node;
 31       node->prev = new_node;
 32       return 1;
 33 }
 34
 35 void list_remove(List_node *node) {
 36       node->next->prev = node->prev;
 37       node->prev->next = node->next;
 38       free(node);
 39 }
 40
 41 List_node *list_search(List_node *node, void *data,
```

```
 42                          int (*comp) (const void*, const void*)) {
 43        List_node *p;
 44        for (p = node->next; p != node && comp(p->data, data) > 0; p = p->next)
 45              ;
 46        return p;
 47  }
 48
 49  void list_foreach(List_node *node, void (*do_it)(void*)) {
 50        for (List_node *p = node->next; p != node; p = p->next)
 51              do_it(p->data);
 52  }
 53
 54  List_node *list_front(List_node *node) {
 55        return node->next;
 56  }
 57
 58  void *list_data(List_node *node) {
 59        return node->data;
 60  }
 61
 62  int list_empty(List_node *node) {
 63        return node->next == node;
 64  }
```

listd.c

Para produzir o programa final:

```
$ gcc wqueue.c list_d.c -o wqueue
```

5 Lista duplamente ligada genérica intrusiva em separado

A utilização de lista intrusiva tem vantagens em relação à utilização da lista não intrusiva. Na lista não intrusiva é necessário o processamento adicional de criação do nó da lista e o acesso ao campo de dados implica uma indireção.

Como característico de uma lista intrusiva os elementos de formação da lista são integrados na própria *struct* de dados. Nesta implementação esses elementos são campo **node** do tipo **List_node**. Os dados de utilizador são os mesmos.

```
typedef struct User {
      time_t arrival;
      List_node node;
      char *name;
} User;
```

O tipo **List_node** é composto apenas pelos ponteiros de ligação.

```
typedef struct List_node {
      struct List_node *next, *prev;
} List_node;
```

A lista é representada por um ponteiro para o elemento sentinela, do tipo **List_node**.

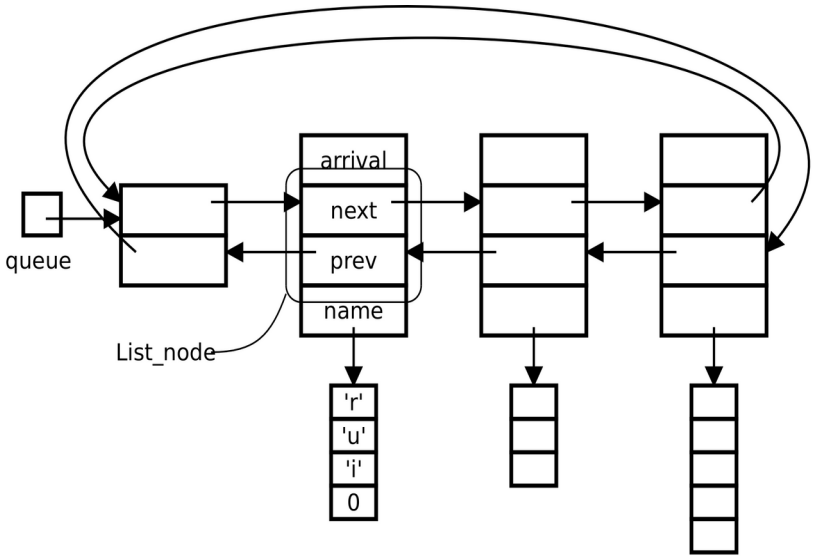
```
List_node *queue;
```

Antes de utilizar a lista é preciso iniciar o ponteiro **queue**.

```
queue = list_create();
```

A diferença entre a utilização de lista não intrusiva ou de lista intrusiva está na forma de aceder aos dados de utilizador – na primeira é através do ponteiro para os dados existente no próprio nó da lista, e na segunda é por ajuste do ponteiro para o nó da lista integrado na própria *struct* do utilizador.

Na implementação da lista genérica não intrusiva a referência para os dados era feita através de um ponteiro para a struct User. Nesta implementação a referência para is dados é feita de forma indireta através do ponteiro para o nó.



Inserir um novo utente na fila

A referência para inserção de uma **`struct User`** na lista é o ponteiro para o campo **`node`** dessa *struct* – linha 7.

```
1 static void user_insert(char * name) {
2       User * user = malloc(sizeof *user);
3       user->name = malloc(strlen(name));
4       strcpy(user->name, name);
5       time(&user->arrival);
6
7       list_insert_rear(queue, &user->node);
8 }
```

Remover utente da fila por atendimento normal

O primeiro elemento da lista, do tipo **`struct User`**, é obtido através da função **`list_remove_front`** – linha 4. A referência recebida é um ponteiro para o campo **`node`**, não o ponteiro para o início da **`struct User`**.

Como obter esse ponteiro a partir do ponteiro para o campo **`node`**? Subtraindo a este ponteiro a distância do campo **`node`** ao início da **`struct User`** – linha 5.

A macro *offsetof*, definida em *stddef.h*, devolve a distância do campo indicado até ao início da *struct* a que o campo pertence. **`offsetof(User, node)`** devolve a distância do campo **`node`** ao início de **`User`**.

```
1  char *user_answer() {
2        if (list_empty(queue))
3              return NULL;
4        List_node *node = list_remove_front(queue);
5        User *user = (User*)((char*)node - offsetof(User, node));
6
7        char *name = user->name;
8        free(user);
9        return name;
10 }
```

Remover utente da fila por desistência

```
1  static int cmp(List_node *node, const void *name) {
2        User *user = (User*)((char*)node - offsetof(User, node));
3        return strcmp(user->name, name);
4  }
5
6  static void user_remove(char * name) {
7        if (list_empty(queue))
8              return;
9        List_node * node = list_search(queue, name, cmp);
10       if (node == NULL)
11             return;
12       User *user = (User*)((char*)node - offsetof(User, node));
13       list_remove(node);
14       free(user->name);
15       free(user);
16 }
```

Listar utentes na fila

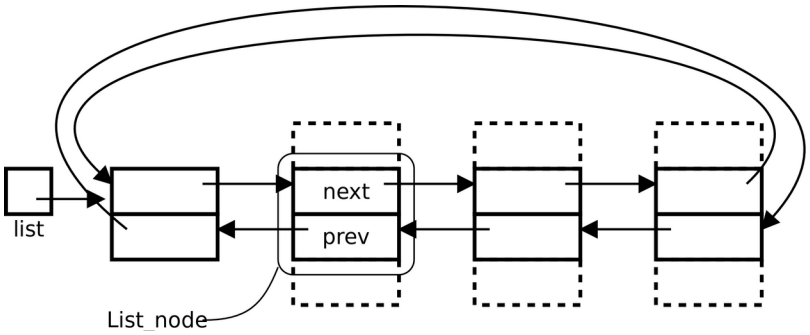
```
1  void print(List_node *node) {
2        int n = 1;
3        User *user = (User*)((char*)node - offsetof(User, node));
4        printf("%d: %s\n", n++, user->name);
5  }
6
7  void user_print() {
8        if (list_empty(queue)) {
9              printf("Fila vazia\n");
10             return;
11       }
12       list_foreach(queue, print);
13 }
```

Vazar fila de utentes

```
1  void free_user(List_node *node) {
2        User *user = (User*)((char*)node - offsetof(User, node));
3        free(user->name);
4        free(user);
5  }
6
7  void user_remove_queue() {
8        list_foreach(queue, free_user);
```

```
    9       list_destroy(queue);
   10 }
```

Lista genérica; dupla ligação; intrusiva – implementação



listdi.h

```
    1 #ifndef LISTDI_H
    2 #define LISTDI_H
    3
    4 typedef struct List_node {
    5       struct List_node *next, *prev;
    6 } List_node;
    7
    8 List_node *list_create();
    9
   10 void list_destroy(List_node *list);
   11
   12 void list_insert_rear(List_node *list, List_node *new_node);
   13
   14 void list_remove(List_node *node);
   15
   16 int list_empty(List_node *list);
   17
   18 List_node * list_front(List_node *list);
   19
   20 List_node * list_search(List_node *list, const void *key,
   21                         int (*cmp)(List_node *, const void *));
   22
   23 void list_foreach(List_node *list, void (*do_it)(List_node *));
   24
   25 #endif
```

listdi.h

listdi.c

```
    1 #include <stdlib.h>
    2 #include <assert.h>
    3 #include "list_di.h"
    4
    5 void list_init(List_node *node) {
    6       node->next = node->prev = node;
    7 }
    8
    9 List_node *list_create() {
   10       List_node *node = malloc(sizeof(List_node));
   11       if (node == 0)
   12             return 0;
```

```
 13       list_init(node);
 14       return node;
 15 }
 16
 17 void list_destroy(List_node *list) {
 18       assert(list);
 19       free(list);
 20 }
 21
 22 void list_insert_rear(List_node *list, List_node *new_node) {
 23       assert(list && new_node);
 24       new_node->prev = list->prev;
 25       new_node->next = list;
 26       list->prev->next = new_node;
 27       list->prev = new_node;
 28 }
 29
 30 void list_remove(List_node *node) {
 31       assert(node);
 32       if (list_empty(node))
 33             return;
 34       node->prev->next = node->next;
 35       node->next->prev = node->prev;
 36 }
 37
 38 int list_empty(List_node *list) {
 39       assert(list);
 40       return list->next == node;
 41 }
 42
 43 List_node *list_front(List_node *list) {
 44       assert(list);
 45       return list->next;
 46 }
 47
 48 List_node *list_search(List_node *list, const void *key,
 49                        int (*cmp)(List_node *, const void *)) {
 50       for (List_node *p = list->next; p != list; p = p->next)
 51             if (cmp(p, key) == 0)
 52                   return p;
 53       return NULL;
 54 }
 55
 56 void list_foreach(List_node *list, void (*opr)(List_node *)) {
 57       for (List_node *next, *p = list->next; p != list; p = next) {
 58             next = p->next;
 59             opr(p);
 60       }
 61 }
```

listdi.c

Nos exemplos seguintes é utilizado um programa contador de palavras para ilustrar a utilização de árvore binária, vetor e tabela de dispersão. São realizadas três versões, cada uma com uma estrutura de dados diferente como contentor de palavras. As operações sobre o contentor mais realizadas neste programa são a pesquisa por palavra, seguida da inserção, caso ainda não exista.

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <time.h>
 5
 6 unsigned int get_time() {
 7       struct timespec ts;
 8       clock_gettime(CLOCK_MONOTONIC, &ts);
 9       return ts.tv_sec * 1000 + (ts.tv_nsec / 1000000);
10 }
11
12 char *separators = " .,;!?\t\n\f:-\"\'(){}[]*=%><#+-&";
13
14 char *word_read(FILE *fd){
15       static char buffer[100];
16       int i = 0, c;
17       do {
18             c = fgetc(fd);
19             if (c == EOF)
20                   return NULL;
21       } while (strchr(separators, c) != NULL);
22       do {
23             buffer[i++] = c;
24             c = fgetc(fd);
25             if(c == EOF)
26                   break;
27       } while (strchr(separators, c) == NULL);
28       buffer[i] = '\0';
29       return buffer;
30 }
31
32 typedef struct Word {
33       char *text;
34       int counter;
35 } Word;
36
37 ... *words;       /* ponteiro para o contentor */
38
39 int main(int argc, char *argv[]){
40       FILE *fd = fopen(argv[1], "r");
41       if (NULL == fd) {
42             fprintf(stderr, "fopen(%s, \"r\"): %s\n",
43                   argv[1], strerror(errno));
44             exit(-1);
45       }
46       int nwords = 0;
47       words = ...   /*  inicializar contentor */
48
49       long initial = get_time();
50       char *word_text = word_read(fd);
51       while (word_text != NULL) {
52             nwords++;
```

```
53              /*
54              if (palavra já existe no contentor?)
55                    incrementar contador da palavra
56              else
57                    inserir nova palavras no contentor
58              */
59              word_text = word_read(fd);
60        }
61        long duration = get_time() - initial;
62        printf("Total de palavras = %d; "
63                  "Palavras diferentes = %ld Time = %ld\n",
64                   nwords, vector_size(words), duration);
65        fclose(fd);
66        /* vazar o contentor de palavras em seguida eliminar o contentor */
67  }
68
```

Árvore binária

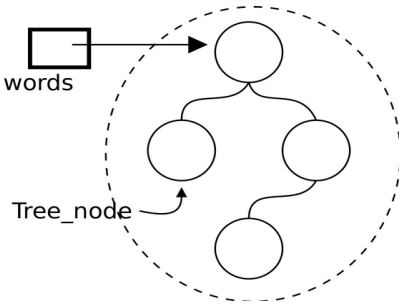
Uma árvore binária suporta uma coleção ordenada de elementos de dados. No programa contador de palavras os elementos de dados são a **`struct Word`**, a chave de ordenação é a própria palavra.

É utilizada uma implementação genérica de árvore binária não balanceada, cujos nós são objetos do tipo **`struct Tree_node`**, nos quais o campo de dados aponta uma **`struct Word`**.

```
typedef struct Tree_node {
      struct Tree_node *left, *right;
      void *data;
} Tree_node;
```

Uma árvore é representada por um ponteiro para **`Tree_node`**, que deve ser inicializado com **`NULL`** para representar uma árvore vazia.

```
Tree_node *words = NULL;
```



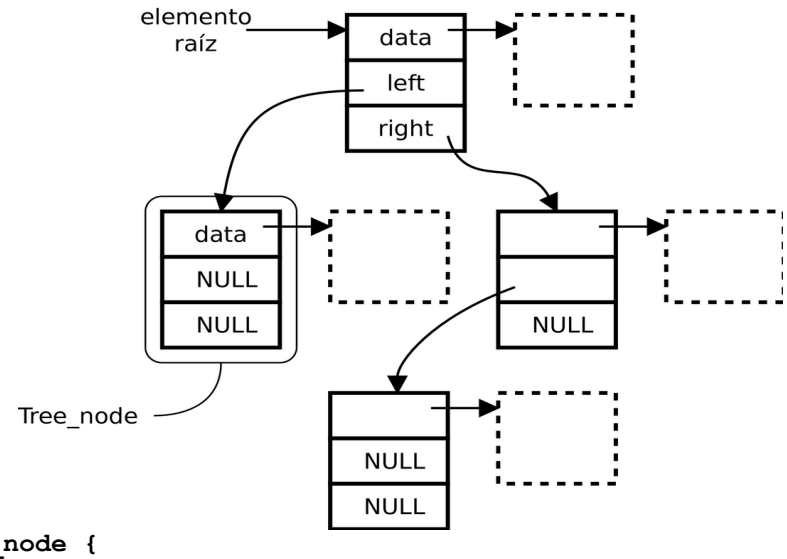
A operação de procura da palavra é realizada pela função **`tree_search`** – linha 24. Esta função recebe a palavra a procurar e a função de comparação e devolve o nó da árvore onde a palavra está alojada. Caso a palavra ainda não exista é devolvido **`NULL`**, ao que se segue a inserção de nova palavra com a função **`tree_insert`** – linha 33.

Ambas as funções recebem a função de comparação **`word_cmp_text`** que compara a palavra presente num nó da árvore, acessível pelo parâmetro **`a`**, com uma segunda palavra. acessível pelo parâmetro **`b`**.

```
 1 int word_cmp_text(void *a, void *b) {
 2       return strcmp(((Word *)a)->text, ((Word*)b)->text);
 3 }
 4
 5 void free_word(void *data) {
 6       free(((Word *)data)->text);
 7       free(data);
 8 }
 9
10 int main(int argc, char *argv[]) {
11       FILE *fd = fopen(argv[1], "r");
12       if (NULL == fd) {
13             fprintf(stderr, "fopen(%s, \"r\"): %s\n",
14                         argv[1], strerror(errno));
15             exit(-1);
16       }
17       int nwords = 0;
18       words = NULL;
19       long initial = get_time();
20       char *word_text = word_read(fd);
21       while(word_text != NULL) {
22             nwords++;
23             Word key = {.text = word_text};
24             Tree_node *node = tree_search(words, &key, word_cmp_text);
25             if (node != NULL) {
```

```
26                  Word *word = tree_data(node);
27                  word->counter++;
28            }
29            else {
30                  Word *word = malloc(sizeof(Word));
31                  word->counter = 1;
32                  word->text = strdup(word_text);
33                  words = tree_insert(words, word, word_cmp_text);
34            }
35            word_text = word_read(fd);
36      }
37      long duration = get_time() - initial;
38      printf("Total de palavras = %d; "
39                  "Palavras diferentes = %ld Time = %ld\n",
40            nwords, tree_size(words), duration);
41      printf("Profundidade da árvore = %d\n", tree_depth(words));
42      /* libertar a memória alocada */
43      fclose(fd);
44      tree_foreach(words, free_word);
45      tree_destroy(words);
46 }
```

1 Árvore binária genérica – implementação




```
 1 typedef struct Tree_node {
 2       struct Tree_node *left, *right;
 3       void *data;
 4 } Tree_node;
 5
 6 void tree_destroy(Tree_node *node) {
 7       if (node->left != NULL)
 8             tree_destroy(node->left);
 9       if (node->right != NULL)
10             tree_destroy(node->right);
11       free(node);
12 }
13
14 Tree_node *tree_search(Tree_node *node, void *data,
15                        int(*cmp)(void*, void*)) {
16       if (NULL == node)
```

```
17              return NULL;
18        int cmp_result = cmp(node->data, data);
19        if (cmp_result == 0)
20              return node;
21        if (cmp_result < 0)
22              return tree_search(node->right, data, cmp);
23        else
24              return tree_search(node->left, data, cmp);
25  }
26
27  Tree_node *tree_insert(Tree_node *node, void *data,
28                                        int (*cmp)(void*, void*)) {
29        if (NULL == node) {
30              node = malloc(sizeof *node);
31              if (NULL == node)
32                    return NULL;
33              node->left = node->right = NULL;
34              node->data = data;
35        }
36        else if (cmp(node->data, data) < 0)
37              node->right = tree_insert(node->right, data, cmp);
38        else
39              node->left = tree_insert(node->left, data, cmp);
40        return node;
41  }
42
43  size_t tree_size(Tree_node *node)  {
44        if (NULL == node)
45              return 0;
46        return tree_size(node->left) + tree_size(node->right) + 1;
47  }
48
49  void tree_foreach(Tree_node *node, void (*do_it)(void*)) {
50        if (NULL == node)
51              return;
52        tree_foreach(node->left, do_it);
53        do_it(node->data);
54        tree_foreach(node->right, do_it);
55  }
56
57  static int max(int a, int b) {
58        return (a > b) ? a : b;
59  }
60
61  int tree_depth(Tree_node *node) {
62        if (node == NULL)
63              return 0;
64        return max(tree_depth(node->left), tree_depth(node->right)) + 1;
65  }
66
67  void *tree_data(Tree_node *node) {
68        return node->data;
69  }
```

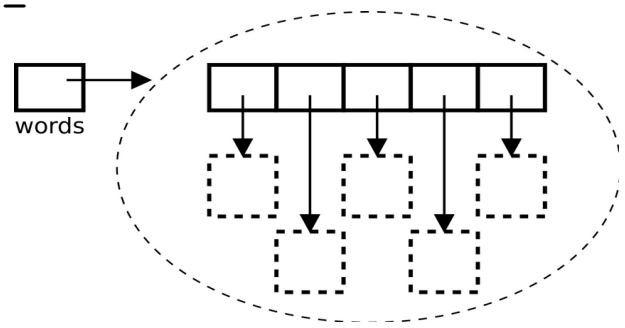
2 Vetor

A estrutura de dados vetor tem como principais vantagens o acesso direto aos elementos a custo unitário e quando ordenado permite pesquisa dicotómica a custo log n.

Neste exemplo apresenta-se a utilização e a implementação um vetor de ponteiros dinâmico, isto é, os elementos que armazena são ponteiros indiferenciados e cresce à medida que se vão introduzindo mais elementos.

A estrutura de suporte **`struct Vector`** é opaca para o utilizador, só deve ser utilizada através das funções de manipulação do vetor. O contentor é representada por um ponteiro para **`Vector`**, que deve ser inicializado com o retorno da função **`vector_create`**.

```
Vector *words;
```

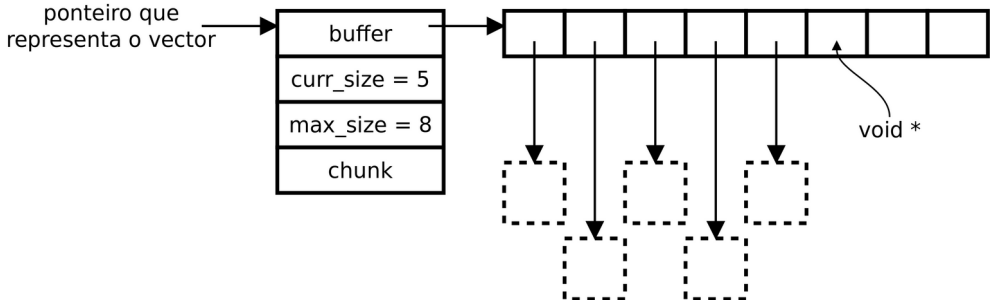


A operação de procura de palavra é realizada pela função **`vector_sorted_search`** – linha 10. Esta função recebe a palavra a procurar – **`word_text`**, a função de comparação – **`word_cmp_text`**, o ponteiro para receber um índice do vetor – **`&index`** e devolve uma indicação booleana sobre se a palavra foi encontrada. O índice indica a posição do vetor onde a palavra se encontra ou caso não tenha sido encontrada a posição onde deve ser inserida. Esta informação de posição fica disponível depois da procura e é aproveitada para a inserção de nova palavra através da função **`vector_insert`** – linha 19.

```
 1 int word_cmp_text(const void *a, const void *b) {
 2       return strcmp((*(Word **)a)->text, (const char *)b);
 3 }
 4
 5 void free_words(void *data) {
 6       free(((Word *)data)->text);
 7       free(data);
 8 }
 9
10 int main(int argc, char *argv[]){
11       FILE *fd = fopen(argv[1], "r");
12       if (NULL == fd) {
13             fprintf(stderr, "fopen(%s, \"r\"): %s\n",
14                         argv[1], strerror(errno));
15             exit(-1);
16       }
17       int nwords = 0;
18       words = vector_create(1000);
19       long initial = get_time();
20       char *word_text = word_read(fd);
21       while (word_text != NULL) {
22             nwords++;
23             size_t index;
24             if (vector_sorted_search(words, word_text,
25                                                 word_cmp_text, &index)) {
26                   Word *w = vector_at(words, index);
27                   w->counter++;
28             }
```

```
29              else {
30                    Word *w = malloc(sizeof(Word));
31                    w->counter = 1;
32                    w->text = strdup(word_text);
33                    vector_insert(words, w, index);
34              }
35              word_text = word_read(fd);
36        }
37        long duration = get_time() - initial;
38        fclose(fd);
39        printf("Total de palavras = %d; "
40                  "Palavras diferentes = %ld Time = %ld\n",
41                   nwords, vector_size(words), duration);
42        vector_foreach(words, free_words);
43        vector_destroy(words);
44  }
```

Vetor de ponteiros genéricos – implementação




```
 1  typedef struct Vector {
 2        void **buffer;
 3        size_t current_size, max_size, chunk;
 4  } Vector;
 5
 6  Vector *vector_create(int chunk) {
 7        Vector *vector = malloc(sizeof *vector);
 8        if (NULL == vector)
 9              return NULL;
10        vector->buffer = malloc(chunk * sizeof *vector->buffer);
11        if (NULL == vector->buffer) {
12              free(vector);
13              return NULL;
14        }
15        vector->chunk = chunk;
16        vector->max_size = chunk;
17        vector->current_size = 0;
18        return vector;
19  }
20
21  void vector_destroy(Vector *vector) {
22        free(vector->buffer);
23        free(vector);
24  }
25
26  size_t vector_sorted_search(Vector *vector, void *key,
27                        int (*cmp)(const void *, const void *), size_t *result) {
28        size_t left = 0, right = vector->current_size;
```

```
29       size_t middle = (right - left) / 2;
30       size_t ref = 0;
31       void **buffer = vector->buffer;
32       while (left < right) {
33             int cmp_result = cmp(&buffer[middle], key);
34             if (cmp_result < 0)
35                   ref = left = middle + 1;
36             else if (cmp_result > 0)
37                   ref = right = middle;
38             else {
39                   *result = middle;
40                   return 1;
41             }
42             middle = left + (right - left) / 2;
43       }
44       *result = ref;
45       return 0;
46 }
47
48 size_t vector_size(Vector * vector) {
49       return vector->current_size;
50 }
51
52 void *vector_at(Vector * vector, int index) {
53       return vector->buffer[index];
54 }
55
56 int vector_insert(Vector *vector, void *element, size_t index) {
57       void **buffer = vector->buffer;
58       if (vector->current_size == vector->max_size) {
59             vector->buffer = buffer = realloc(buffer,
60                   (vector->current_size + vector->chunk) * sizeof *buffer);
61             if (NULL == buffer)
62                   return 0;
63             vector->max_size += vector->chunk;
64       }
65       memmove(&buffer[index + 1], &buffer[index],
66                   (vector->current_size - index) * sizeof *buffer);
67       buffer[index] = element;
68       vector->current_size++;
69       return 1;
70 }
71
72 void vector_remove(Vector *vector, size_t index) {
73       memmove(&vector->buffer[index], &vector->buffer[index + 1],
74             (vector->current_size - index - 1) * sizeof *vector->buffer);
75       vector->current_size--;
76 }
77
78 void vector_sort(Vector *vector, int (*cmp)( const void *, const void *)) {
79     qsort(vector->buffer, vector->current_size, sizeof *vector->buffer, cmp);
80 }
81
82 void vector_foreach(Vector * vector, void(*do_it)(void *)) {
83       for (size_t i = 0; i < vector->current_size; ++i)
84             do_it(vector->buffer[i]);
85 }
```

3 Tabela de dispersão (*hash table*)

A estrutura de dados tabela de dispersão tem como principal vantagem a procura de elementos – na presença de poucas colisões tem custo unitário.

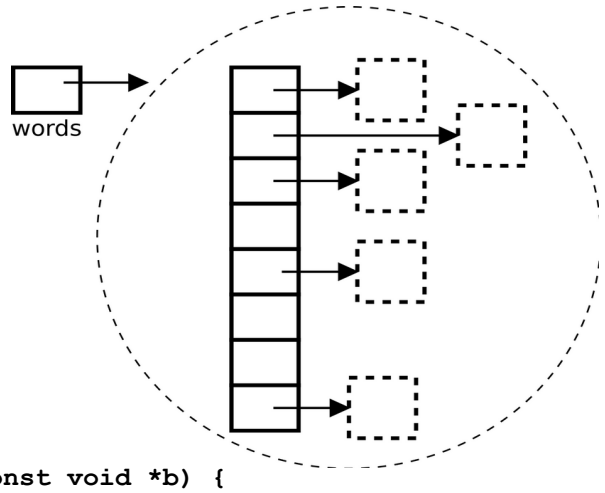
Neste exemplo apresenta-se a utilização e a implementação de uma tabela de dispersão que armazena ponteiros indiferenciados com colisões resolvidas por lista simplesmente ligada.

A estrutura de suporte **`struct Htable`** é opaca para o utilizador, só deve ser utilizada através das funções de manipulação da tabela. Uma tabela é representada por um ponteiro para **`Htable`**, que deve ser inicializado com o retorno da função **`htable_create`**.

```
Htable *words;

words = htable_create(10000, hash_function, word_cmp_text);
```

A função **`htable_create`** recebe como argumentos a dimensão da tabela, a função de *hash* e a função de comparação da chave. Da dimensão da tabela e a da qualidade da função de *hash* depende a taxa de colisões.

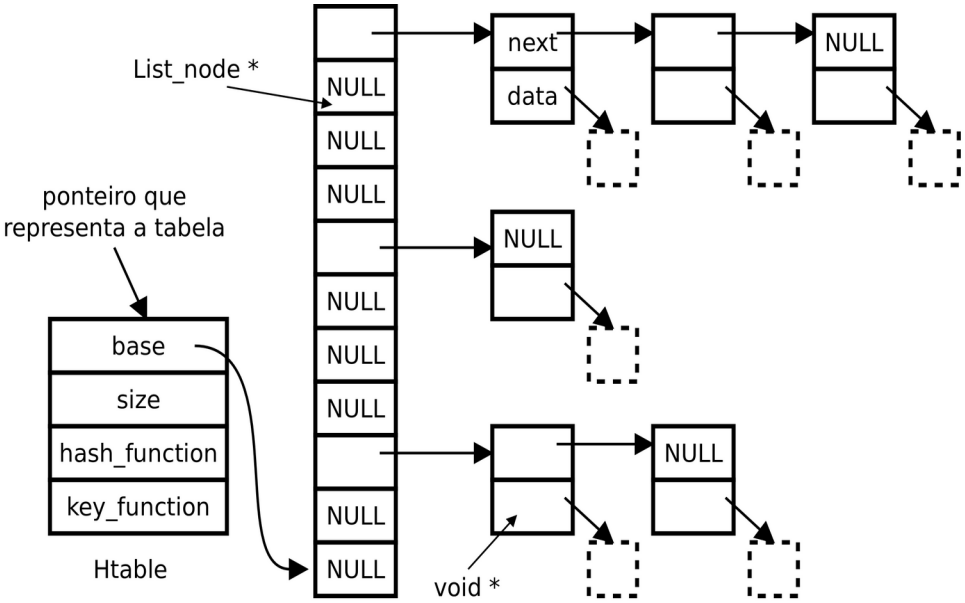


A operação de procura de palavra é realizada pela função **`htable_lookup`** – linha 31. Esta função recebe a palavra a procurar e devolve o ponteiro para o elemento de dados encontrado. Se devolver **`NULL`** significa que a palavra procurada não se encontra na tabela.

```
 1 int word_cmp_text(const void *data, const void *b) {
 2       return strcmp(((Word *)data)->text, (char*)b);
 3 }
 4
 5 int hash_function(const void *data) {
 6       const char *word_text = data;
 7       int hash = 0;
 8       while (*word_text)
 9             hash += *word_text++;
10       return hash;
11 }
12
13 void free_words(void *data) {
14       free(((Word *)data)->text);
15       free(data);
16 }
17
18 int main(int argc, char *argv[]){
19       FILE *fd = fopen(argv[1], "r");
20       if (NULL == fd) {
21             fprintf(stderr, "fopen(%s, \"r\"): %s\n",
22                         argv[1], strerror(errno));
23             exit(-1);
24       }
25       int nwords = 0;
26       words = htable_create(10000, hash_function, word_cmp_text);
```

```
27        long initial = get_time();
28        char *word_text = word_read(fd);
29        while(word_text != NULL) {
30              nwords++;
31              Word *word = htable_lookup(words, word_text);
32              if (word != NULL) {
33                    word->counter++;
34              }
35              else {
36                    word = malloc(sizeof(Word));
37                    word->counter = 1;
38                    word->text = strdup(word_text);
39                    htable_insert(words, word_text, word);
40              }
41              word_text = word_read(fd);
42        }
43        long duration = get_time() - initial;
44        printf("Total de palavras = %d; "
45                    "Palavras diferentes = %ld Time = %ld\n",
46                    nwords, htable_size(htable), duration);
47        printf("Collisions = %d\n", htable_collisions(htable));
48        fclose(fd);
49        htable_foreach(words, free_words);
50        htable_destroy(words);
51  }
```

Tabela de dispersão; ponteiros genéricos – implementação




```
 1 typedef struct Htable {
 2       List_node **base;
 3       size_t size;
 4       int (*hash_function)(const void *);
 5       int (*key_function)(const void *, const void *);
 6 } Htable;
 7
 8 void htable_init(Htable * htable, size_t size,
 9                         int (*hf)(const void *),
10                         int (*kf)(const void *, const void *)) {
11       htable->hash_function = hf;
12       htable->key_function = kf;
13       htable->size = size;
14       for (size_t i = 0; i < size; ++i)
15             htable->base[i] = NULL;
16 }
17
18 Htable *htable_create(size_t size,
19                         int (*hf)(const void *),
20                         int (*kf)(const void *, const void *)) {
21       Htable *htable = malloc(sizeof(Htable));
22       if (NULL == htable)
23             return NULL;
24       htable->base = malloc(sizeof *htable->base * size);
25       if (NULL == htable->base){
26             free(htable);
27             return NULL;
28       }
29       htable_init(htable, size, hf, kf);
30       return htable;
31 }
32
33 void htable_destroy(Htable *htable) {
34       for (size_t i = 0; i < htable->size; ++i)
35             if (htable->base[i] != NULL)
```

```
36                      list_destroy(htable->base[i]);
37          free(htable->base);
38          free(htable);
39  }
40
41  void htable_insert(Htable *htable, const void *key, void *data) {
42          int index = htable->hash_function(key) % htable->size;
43          htable->base[index] = list_insert(htable->base[index], data);
44  }
45
46  void *htable_lookup(Htable *htable, const void *key) {
47          int index = htable->hash_function(key) % htable->size;
48          if (NULL == htable->base[index])
49                  return NULL;
50          List_node *node = list_search(htable->base[index],
51                                        key, htable->key_function);
52          if (node != NULL)
53                  return list_data(node);
54          return NULL;
55  }
56
57  void htable_foreach(Htable *htable, void (*do_it)(void*)) {
58          for (size_t i = 0; i < htable->size; ++i)
59                  if (htable->base[i] != NULL)
60                          list_foreach(htable->base[i], do_it);
61  }
62
63  size_t htable_size(Htable *htable) {
64          int i, counter = 0;
65          for (i = 0; i < htable->size; ++i)
66                  if (NULL != htable->base[i])
67                          counter += list_size(htable->base[i]);
68          return counter;
69  }
```