


Estruturas de Dados / Programação 2
Listas Encadeadas

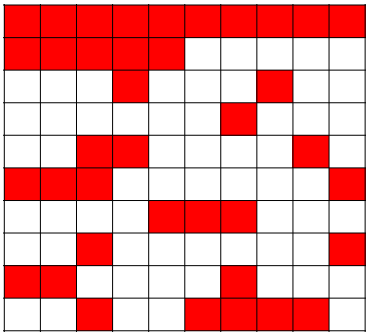
Márcio Ribeiro
marcio@ic.ufal.br
twitter.com/marciomribeiro

Arrays

- What are the advantages of arrays?
- What are the disadvantages of arrays?
- Please, take 3 minutes to discuss with your friend about the above questions

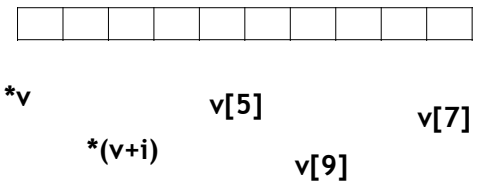
Let's take a look on the memory allocation...

- Can we create an array of 10 positions?



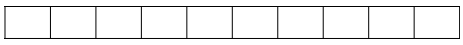
Unfortunately we can't...

- Arrays require contiguous positions!
- That's why we can access any position randomly!



What happens if we access a non-valid position?

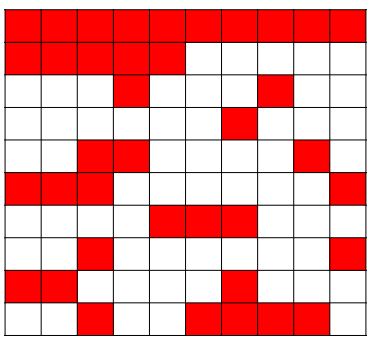
- It depends on the language...
  - In C, some *unknown* value is recovered
  - In Java, an *error* is raised (the bounds are checked!)



a[119] = ?

Imagine that we MUST use an array with 10 positions!

- Is there anything in C language that can help us?



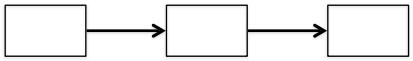
Linked List

Introduction

- You probably mentioned **pointers**
- And you are right!
- Now, we will study a linear data structure (just like the known arrays) named **Linked Lists**

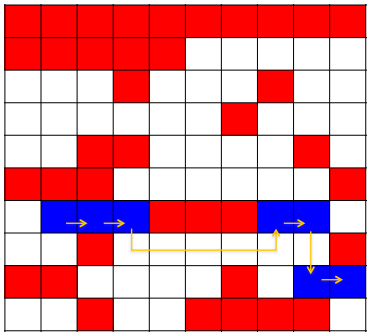
Linked Lists

- Pointers are used to link each node of our list
- We traverse the list by using the pointers
- Rectangle = Node
- Arrow = Pointer



No need to be continuous… Can't do *(v+i)

- We cannot guarantee that it will be continuous!



Linked Lists are Dynamic Data Structures! They grow as we want!

This way, there is no direct access!

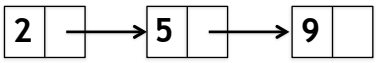
- We navigate throughout the list by using the pointers…
- But… where are the data?
- Only pointers were presented so far…
- What each node should store?
- Take 3 minutes to discuss these questions

```
struct node {
  node *next;
};
```

- In this particular case, this struct seems useless!
- Each node should also contain the data we want to manipulate by inserting, removing, searching etc!

Node: data + pointer

```
struct node {
  int item;
  node *next;
};
```



Abstract Data Type: Linked List

Linked List ADT

```
node* create_linked_list();

node* add(node *head, int item);

node* search(node *head, int item);

node* remove(node *head, int item);

int is_empty(node *head);

void print_linked_list(node *head);
```

Let's implement the TAD…

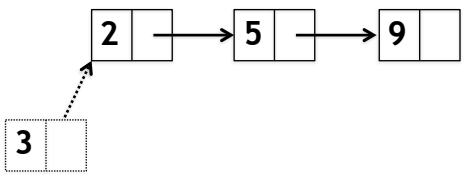
- Creating a Linked List: return a pointer that points to NULL
- This will be useful to navigate throughout the Linked List

```
node* create_linked_list()
{
  return NULL;
}

int is_empty(node *head)
{
  return (head == NULL);
}
```

Adding elements… (at the beginning)

```
node* add(node *head, int item)
{
  node *new_node = (node*) malloc(sizeof(node));
  new_node->item = item;
  new_node->next = head;
  return new_node;
}
```



Client code…

- Inserting 5 elements and then printing the list

```
int main()
{
  node* list = create_linked_list();

  list = add(list, 3);
  list = add(list, 9);
  list = add(list, 27);
  list = add(list, 81);
  list = add(list, 243);

  printf("Complete list: \n");
  print_linked_list(list);
}
```

Exercise 1: write the *print_linked_list* function

```
void print_linked_list(node *head)
{
  while (head != NULL) {
    printf("%d\n", head->item);
    head = head->next;
  }
}
```

Exercise 2: complete the *search* function

```
node* search(node *head, int item)
{
  while (head != NULL) {



  }
  return NULL;
}
```

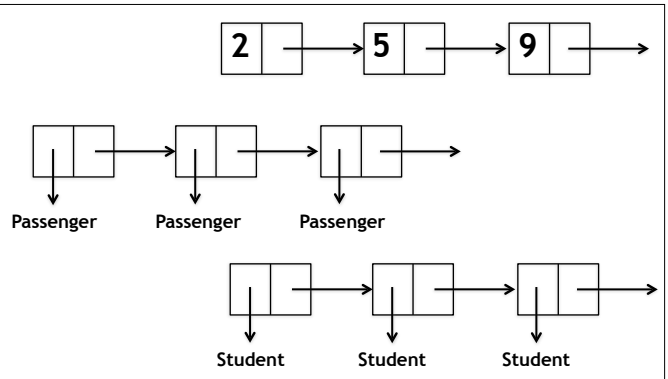
Exercise 3: write the *remove* function

```
node* remove(node *head, int item)
{
  node *previous = NULL;
  node *current = head;
  while (current != NULL && current->item != item) {
    previous = current;
    current = current->next;
  }
  if (current == NULL) {
    return head;
  }
  if (previous == NULL) {
    head = current->next;
  } else {
    previous->next = current->next;
  }
  free(current);
  return head;
}
```

Exercise 4: write the *print_linked_list* function recursively

```
void print_linked_list(node *head)
{
  if (!is_empty(head)) {
    printf("%d\n", head->item);
    print_linked_list(head->next);
  }
}
```

Only integers?! What about a "Generic" List?!

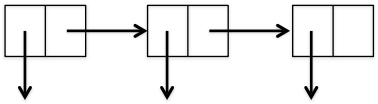



```
struct node {
  ??????????
  node *next;
};
```

```
struct node {
  void *item;
  node *next;
};

void *v;
int *i;
int ivar;
char chvar;
float fvar;
passenger * passenger;

v = &ivar;
v = &chvar;
v = &fvar;
v = &passenger;
i = &ivar;
i = &chvar;
i = &fvar;
```



*void

Can hold the address of any type!
We can assign a pointer of any type to a void pointer!

```
node* search(node *head, void *item)
{
  while (head != NULL) {
    if (head->item == item) {
      return head;
    }
    head = head->next;
  }
  return NULL;
}
```

```
int integers_equals(void *item1, void *item2)
{
  return (*((int*) item1) == *((int*) item2));
}

int strings_equals(void *item1, void *item2)
{
  return !strcmp(item1, item2);
}
```

```
node* search(node *head, void *item,
             int (*equal)(void *item1, void *item2)) {
{
  while (head != NULL) {
    if ((*equal) (head->item, item)) {
      return head;
    }
    head = head->next;
  }
  return NULL;
}

search(list_of_integers, &i, integers_equals);
search(list_of_strings, "IC-UFAL", strings_equals);
search(list_of_passengers, passenger, passengers_equals);
search(list_of_students, student, students_equals);
```

```
void print_linked_list_of_integers(node *head)
{
  while (head != NULL) {
    printf("%d\n", *((int*) head->item));
    head = head->next;
  }
}

void print_linked_list_of_strings(node *head)
{
  while (head != NULL) {
    printf("%s\n", (char*) head->item);
    head = head->next;
  }
}
```

Efficiency

Linked Lists versus Arrays

- Get
  - Array = O(1)
  - Linked List = O(n)
- Insert
  - Array = O(n)
  - Linked List = O(1)
- What can we conclude?

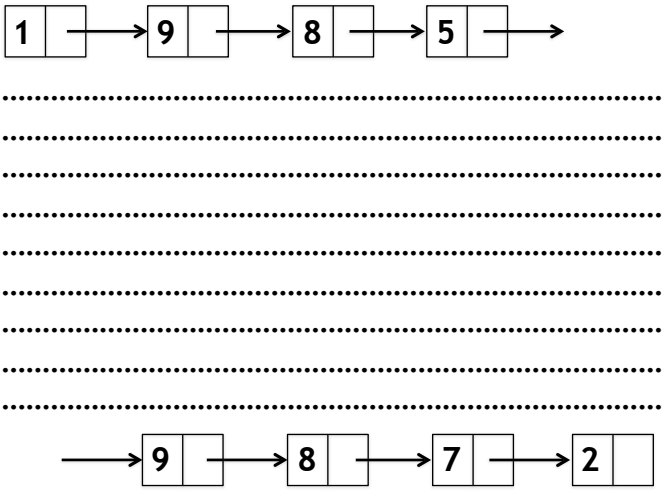
Insert / Delete

- Insert
  - Beginning: O(1)
  - Middle: O(n)
- Delete
  - Beginning: O(1)
  - Middle: O(n)

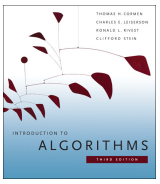
Application

```
19850398512938572385739825698217345892375498723640812640926274823764028365089236590823759032875908365831465831560934875903475918063875698127635863749053419085769813465871682735687236586230198569028317509826395868734650198365982739058690812635876278681763187678612873646839209385689108127364643782823766573829183767832917365738219873657328182735673820128365983260951698356902836590823659082735987 2
```

How to represent?
long long int?



References



Chapter 10

Chapter 3