# Algorithms & Data Structures Notes - SoSe 24

Igor Dimitrov

2024-04-22

# Table of contents

# Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

# Part I

# Introduction

# 1 Program Run-time Analysis

## 1.1 Reccurence Relations

Consider a very simple reccurence relation:

$$T(n) := \begin{cases} 1 & n = 1 \\ n + T(n-1), & n > 1 \end{cases}$$

With **mathematical induction** we can formally show that $T(n)$ is quadratic. But there is a simpler & more intuitive way:

$$\begin{aligned} T(n) &= n + T(n-1) && \text{(Def } T(\cdot)\text{)} \\ &= n + n - 1 + T(n-2) \\ &= ... && \text{(Repeat } n-2 \text{ times)} \\ &= n + n - 1 + \dots + T(1) \\ &= n + n - 1 + \dots + 1 && \text{(Def } T(1)\text{)} \\ &= \frac{n(n+1)}{2} && \text{(Gauss)} \\ &\in \mathcal{O}(n^2) \end{aligned}$$

This method can be applied to the more complex divide-and-conquer reccurence relation from the lecture:

$$R(n) := \begin{cases} a, & n = 1 \\ c\dot{n} + d \cdot R(\frac{n}{b}), & n > 1 \end{cases}$$

Applying the above method we expand $R(\cdot)$ repetitively according to its definition until we reach the base case, rearranging terms when necessary:

$$
\begin{aligned}
R(n) &= c \cdot n + d \cdot R(\frac{n}{b}) && \text{(Def } R(\cdot)\text{)} \\
&= c \cdot n + d(c\frac{n}{b} + d \cdot R(\frac{n}{b^2})) \\
&= c \cdot n + d\Big(c\frac{n}{b} + d \cdot \big(c \cdot \frac{n}{b^2} + d \cdot R(\frac{n}{b^2})\big)\Big) \\
&= c \cdot n + d \cdot c\frac{n}{b} + d^2 c\frac{n}{b^2} + d^3 \cdot R(\frac{n}{b^3}) && \text{(Rearrange)} \\
&= c \cdot n \left(1 + \frac{d}{b} + \frac{d^2}{b^2}\right) + d^3 \cdot R(\frac{n}{b^3}) \\
&= ... && \text{(Repeat } k\text{-times)} \\
&= c \cdot n \left(1 + \frac{d}{b} + \cdots + \frac{d^{k-1}}{b^{k-1}}\right) + d^k \cdot R(\frac{n}{b^k}) \\
&= c \cdot n \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i + d^k \cdot R(\frac{n}{b^k}) \\
&= c \cdot n \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i + d^k \cdot R(1) && \text{(Ass } \tfrac{n}{b^k} = 1\text{)} \\
&= c \cdot n \sum_{i=0}^{k-1} \left(\frac{d}{b}\right)^i + a \cdot d^k && \text{(Def } R(1)\text{)}
\end{aligned}
$$

See lecture slides for the complexity analysis of final expression.

## 1.2 Amortized Analysis

# Part II

# Data Structures

# 2 Lists

## 2.1 Sequences as Arrays and Lists

Many terms for same thing: sequence, field, list, stack, string, **file**… Yes, files are simply sequences of bytes!

three views on lists:

- **abstract**: $(2, 3, 5, 7)$
- **functionality**: stack, queue, etc… What operations does it support?
- **representation**: How is the list represented in a given programming model/language/paradigm?

## 2.2 Applications of Lists

- Storing and processing any kinds of data
- Concrete representation of abstract data types such as: set, graph, etc…

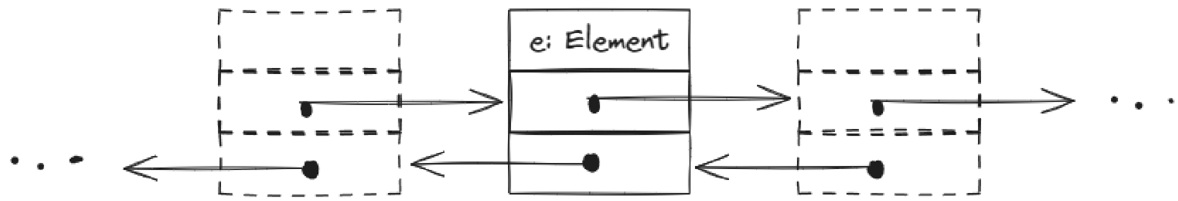
## 2.3 Linked and Doubly Linked Lists

| | simply linked | doubly linked |
|---|---|---|
| lecture | `SList` | `List` |
| c++ | `std::forward_list` | `std::list` |

Doubly linked lists are usually **simpler** and require "only" double the space at most. Therefore their use is more widespread.

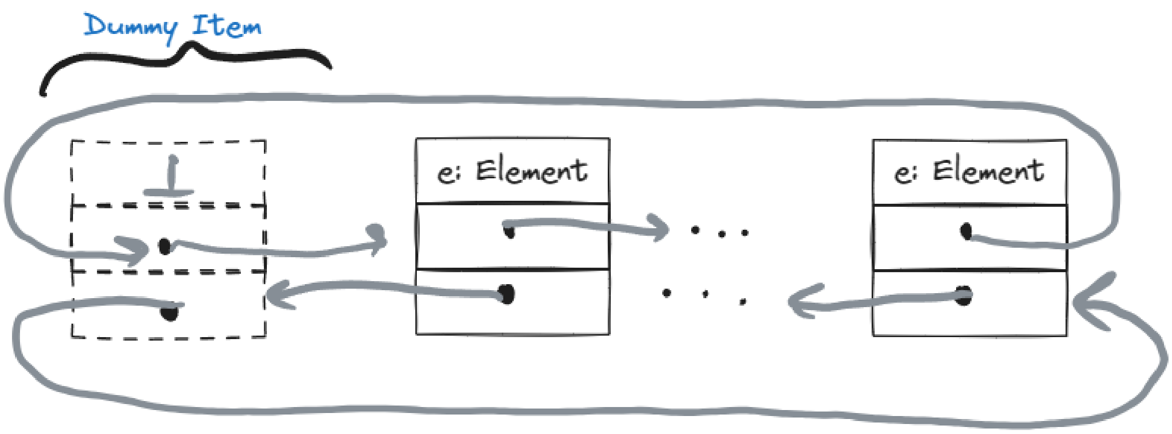
### 2.3.1 List Items

```
Class Item of T :=
    e: T //Data item of type T
    next: *Item //Pointer to Item
    prev: *Item //Pointer to Item
    invariant next->prev = this = prev->next
```




**Problem**: * predeccessor of first list element? * successor of last list element?

**Solution**: Dummy Item with an empty data field as follows:




Advatanges of this solution:

- **Invariant** is always satisfied
- Exceptions are avoided, thus making the coding more:
    - simple
    - readable
    - faster
    - elegant

Disadvantages: a little more storage space.

### 2.3.2 The List Class

```
Class List of T :=
    dummy := (
        Null : T
        &dummy : *T // initially list is empty, therefore next points to the
↪   dummy itself
        &dummy : *T // initially list is empty, therefore prev points to the
↪   dummy itself
    ) : Item

    // returns the address of the dummy, which represents the head of the
↪   list
    Function head() : *Item :=
        return address of dummy

    // simple access functions
    // returns true iff list empty
    Function is_empty() : Bool :=
        return dummy.next == dummy

    // returns pointer to first Item of the list, given list is not empty
    Function first() : *Item :=
        assert (not is_empty())
        return dummy.next

    // returns pointer to last Item of the list, given list is not empty
    Function last() : *Item :=
        assert (not is_empty())
        return dummy.prev

    /* Splice is an all-purpose tool to cut out parts from a list
       Cut out (a, ... b) form this list and insert after t */
    Procedure splice(a, b, t : *Item) :=
        assert (
            b is not before a
            and
            t not between a and b
        )
        // Cut out (a, ... , b)
        a->prev->next := b->next
```

```
    b->next->prev := a->prev

    // insert (a, ... b) after t
    t->next->prev := b
    b->next := t->next
    t->next := a
    a->prev := t


// Moving items by utilising splice
//Move item a after item b
Procedure move_after(a, b: *Item) :=
    splice(a, a, b)

// Move item a to the front of the list
Procedure move_to_front(a: *Item) :=
    move_after(a, dummy)

Procedure move_to_back(a: *Item) :=
    move_after(b, last())

// Deleting items by moving them to a global freeList
// remove item a
Procedure remove(a: *Item) :=
    move_after(b, freeList.dummy)

// remove first item
Procedure pop_front() :=
    remove(first())

//remove last item
Procedure pop_back() :=
    remove(last())

// Inserting Elements
// Insert an item with value x after item a
Function insert_after(x : T, a : *Item) : *Item :=
    checkFreeList() //make sure freeList is non empty
    b := freeList.first() // obtain an item b to hold x
    move_after(b, a) // insert b after a
    b->e := x // set the data item value of b to x
    return b
```

```
// Manipulating whole lists
Procedure concat(L : List) :=
    splice(L.first(), L.last(), last()) //move whole of L after last
↪ element of this list

Procedure clear()
    freeList.concat(this) //after this operation from from first to last
↪ element of this
                          // list are concatenated to the freeList,
↪ leaving only the
                          // dummy element in this list.

Fuction get(i )
```

#### Splicing

The code for splicing of the `List` class:

```
/* Splice is an all-purpose tool to cut out parts from a list
    Cut out (a, ... b) form this list and insert after t */
Procedure splice(a, b, t : *Item) :=
    assert (
        b is not before a
        and
        t not between a and b
    )
    // Cut out (a, ... , b)
    a->prev->next := b->next
    b->next->prev := a->prev

    // insert (a, ... b) after t
    t->next->prev := b
    b->next := t->next
    t->next := a
    a->prev := t
```

- Dlist cut-out $(a, ..., b)$ (see Figure 2.1):

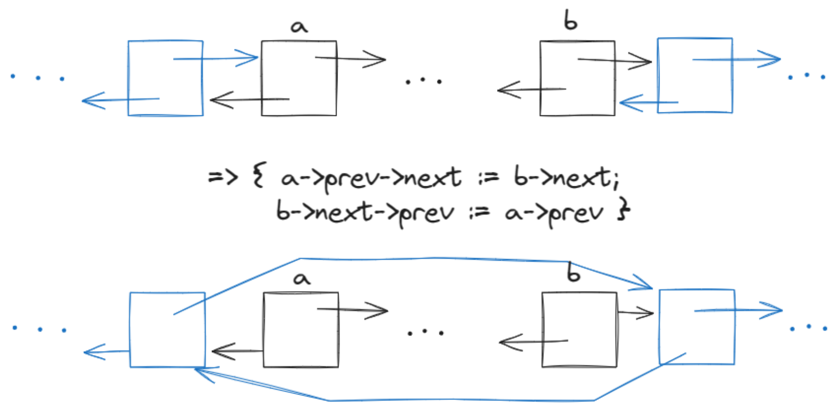



Figure 2.1: cutout

- Dlist insert $(a, ..., b)$ after $t$ (see Figure 2.2):

#### Speicherverwaltung ./.FreeList

Methods (?):

- Niavely: allocate memory for each new element, deallocate memory after deleting each element:
  - advantage: simplicity
  - disadvantage: requires a good implementation of memory management: potentially very slow
- "global" `freeList` (e.g. `static` member in C++)
  - doubly linked list of all not used elements
  - transfer 'deleted' elements in `freeList`.
  - `checkFreeList` allocates, in case the list is empty

Real implementations: * naiv but with well implemented, efficient memory management * refined Free List Approach (class-agnostic, release) * implementation-specific.

#### Deleting Elements

Deleting elements realised by moving them to the global `freeList`:

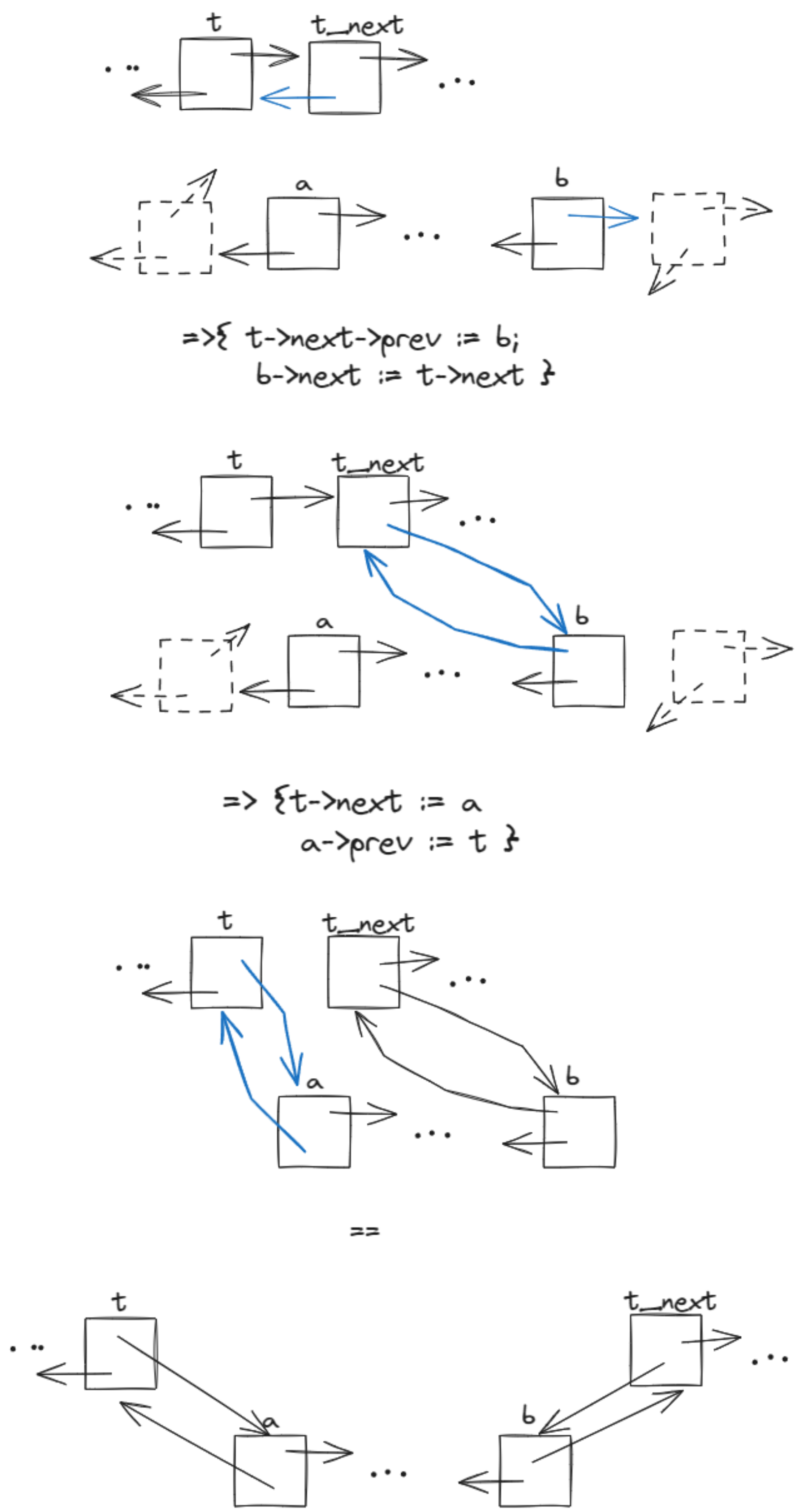



Figure 2.2: insert

```
Procedure remove(a: *Item) :=
    move_after(a, freeList.dummy) // item a is now a 'free' item.

Procedure pop_front() :=
    remove(first())

Procedure pop_back() :=
    remove(last())
```

#### Inserting Elements

Inserting elements into a list $l$ also utilizes `freeList`, by fetching its first element an moving it into $l$.

```
Function insert_after(x : T, a : *Item) : *Item :=
    checkFreeList() //make sure freeList is non empty
    b := freeList.first() // obtain an item b to hold x
    move_after(b, a) // insert b after a
    b->e := x // set the data item value of b to x
    return b

Function insert_before(x : T, b : *Item) : *Item :=
    return insert_after(x, b->prev)

Procedure push_front(x : T) :=
    insert_after(x, dummy)

Procedure push_back(x : T) :=
    insert_after(x, last())
```

#### Manipulating whole Lists

```
// Manipulating whole lists
Procedure concat(L : List) :=
    splice(L.first(), L.last(), last()) //move whole of L after last element
 ↪  of this list

Procedure clear()
```

```
    freeList.concat(this) //after this operation from from first to last
↪   element of this
                            // list are concatenated to the freeList, leaving
↪   only the
                            // dummy element in this list.
```

This operations require **constant time** - indeendent of the list size!

# 3 Arrays

An array is a contigious sequence of memory cells.

## 3.1 Bounded Arrays

Bounded arrays have fixed size and are an efficient data structure.

- Size must be known during compile time and is fixed.
- Its memory location in the stack allows many compiler optimizations.

## 3.2 Unbounded Arrays

The size of an **unbounded array** can dynamically change during run-time. From the user POV it provides the same behaviour as a linked list.

It allows the operations:

- `pushBack(e: T)`: insert an element at the end of the array
- `popBack(e: T)`: remove an element at the end of the array

### 3.2.1 Memory Management

- `allocate(n)`: request a $n$ contigious blocks of memory words and returns the address value of the first block. This we have the memory blocks:

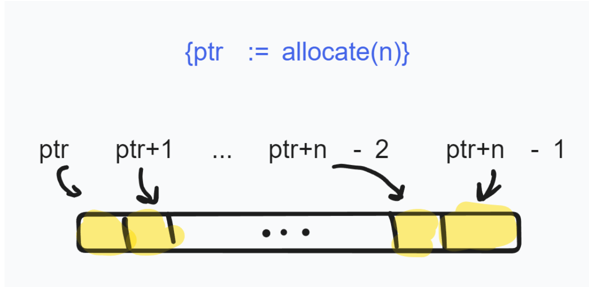



Figure 3.1: array memory allocation

where `ptr + i` addresses are determined by pointer arithmetic.

- `dispose(ptr)` marks the memory address value held in `ptr` as free, effectively deleting the object held there.

In general, the allocated memory can't grow dynamically during life time, since the immediate memory block after the last one might get unpredictably occupied $\Rightarrow$ If we need a new memory block of size $n' > n$, we must allocate a new block, copy the old block contents, and finally free it.

### 3.2.2 Implementation

First we consider a slow variant:

```
Class UArraySlow<T>:=
  c := 0 : Nat // capacity
  b : Array[0..c-1]<T> // the array itself

  pushBack(el : T) : void :=
    // c++
    // allocate new array on heap with new capacity
    // copy elements over from the old array
    // insert el at the last location

  popBack() : void :=
    // analagous
```

**Problem**: $n$ `pushBack` operations require $1 + \dots + n \in \mathcal{O}(n^2)$ time $\Rightarrow$ slow.

Solution:

### Unbounded Arrays with Extra Memory

**Idea**: Request more memory than initial capacity. Reallocate memory only when array gets full or too empty:

**Algorithm design principle**: make common case fast.

```
Class UArray<T> :=
  c := 1 : Nat // capacity
  n := 0 : Nat // number of elements in the array

  //invariant n <= c < k*n || (n == 0 && c < 2)
  b : Array[0..c-1]<T>

  // Array access
  Operator [i : Nat] : T :=
    assert(0 <= i < n)
    return b[i]

  // accessor method for n
  Function size() : Nat := return n

  Procedure pushBack(e : T) :=
    if n == c :
      reallocate(2*n) // see definition below
    b[n] := e
    n++

  // reallocates a new memory with a given capacity c_new
  Procedure reallocate(c_new : Nat) :=
    c := c_new
    b_new := new Array[0..c_new - 1]<T>
    //copy elements over to new array
    for (i = 1 to n - 1) :
      b_new[i] := b[i]
    dispose(b)
    b := b_new

  Procedure popBack() :=
    // don't do anything for empty arrays
    assert n > 0
    n--
```

```
    if 4*n <= c && n > 0 :
      reallocate(2*n)
```