Stacks

Lesson 3

Outline

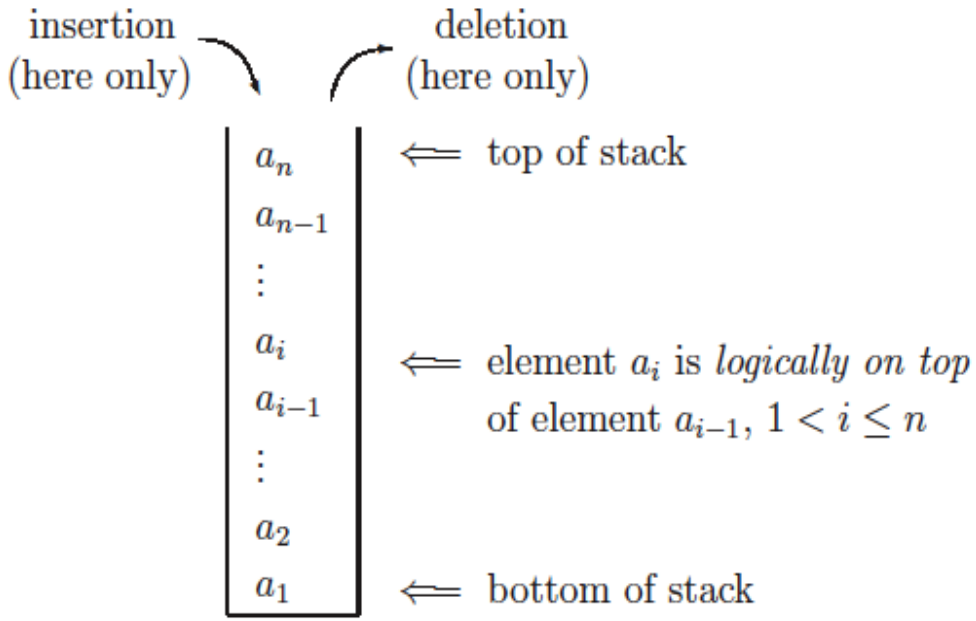
- What is a stack?
- Sequential Implementation
- Linked Implementation
- Application: Pattern recognition
- Multi-stack

Outline

- **What is a stack?**
- Sequential Implementation
- Linked Implementation
- Application: Pattern recognition
- Multi-stack

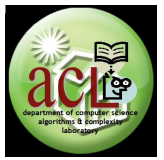
Stack

- ***Last in first out (LIFO)*** container
- Two operations
  - ***Push***
  - ***Pop***
  - ***Auxiliary***
    - ***Initialize stack***
    - ***Emptiness test***
    - ***Fullness test***

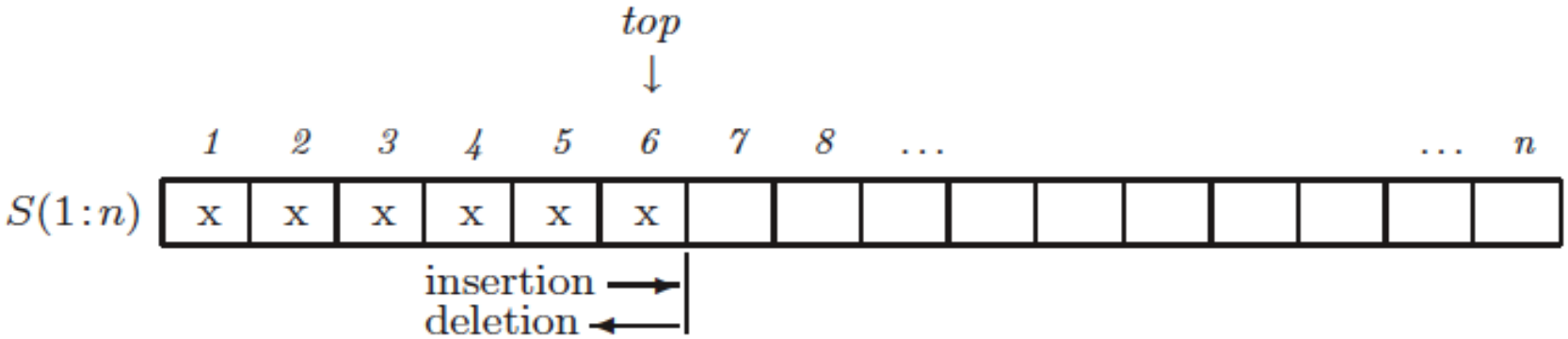


Outline

- What is a stack?
- **Sequential Implementation**
- Linked Implementation
- Application: Pattern recognition
- Multi-stack



An array as a stack



Auxiliary operations

- Initialization

```
procedure InitStack(𝕊)
top ← 0
end InitStack
```

- Emptiness test

```
procedure IsEmptyStack(𝕊)
return(top = 0)
end IsEmptyStack
```

- Fullness test
  - Part of push implementation, although a separate procedure can also be created (how?)

Push and pop

- Push

```
procedure PUSH(𝕊, x)
if top = n then call STACKOVERFLOW
top ← top + 1
S(top) ← x
end PUSH
```

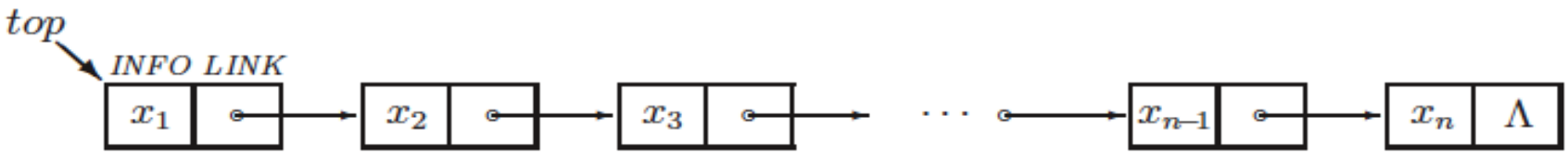
- Pop

```
procedure POP(𝕊, x)
if top = 0 then call STACKUNDERFLOW
           else [x ← S(top); top ← top − 1]
end POP
```

Outline

- What is a stack?
- Sequential Implementation
- **Linked Implementation**
- Application: Pattern recognition
- Multi-stack

A linked list as a stack



Auxiliary operations

- Initialization

```
procedure InitStack(𝕊)
top ← Λ
end InitStack
```

- Emptiness test

```
procedure IsEmptyStack(𝕊)
return(top = Λ)
end IsEmptyStack
```

- Fullness test (How?)

Push and pop

- Push

```
procedure PUSH(𝕊, x)
call GETNODE(α)
INFO(α) ← x
LINK(α) ← top
top ← α
end PUSH
```

- Pop

```
procedure POP(𝕊, x)
if top = Λ then call STACKUNDERFLOW
           else [α ← top
                 x ← INFO(top)
                 top ← LINK(top)
                 call RETNODE(α)]
end POP
```

Outline

- What is a stack?
- Sequential Implementation
- Linked Implementation
- **Application: Pattern recognition**
- Multi-stack

Palindromes

- ***Palindromes*** are words (or clauses) that are the same even when "spelled" backwards
  - e.g. radar, "madam, I'm Adam"
- Create and algorithm that will tell whether a given string is a palindrome or not
  - For simplicity, let's limit this to strings with no spaces and punctuations (except white spaces
  - Also, limit letters to {a, b, c}
    - Furthermore, let c always be the indicator of the middle part of the string

Algorithm pseudocode

1. [Process first half] Get next character, say *char* from input string. If *char* is 'a' or 'b', push *char* onto stack and repeat this step. If *char* is 'c', go to step 2. If *char* is the blank character, return **false** (no second half).
2. [Process second half] Get next character, say *char*, from input string. Pop top element of stack, say x. If $char = x$, repeat this step (OK so far). If $char \neq x$, return **false**. If *char* is the blank character and stack is empty, return **true**; else, return **false**.

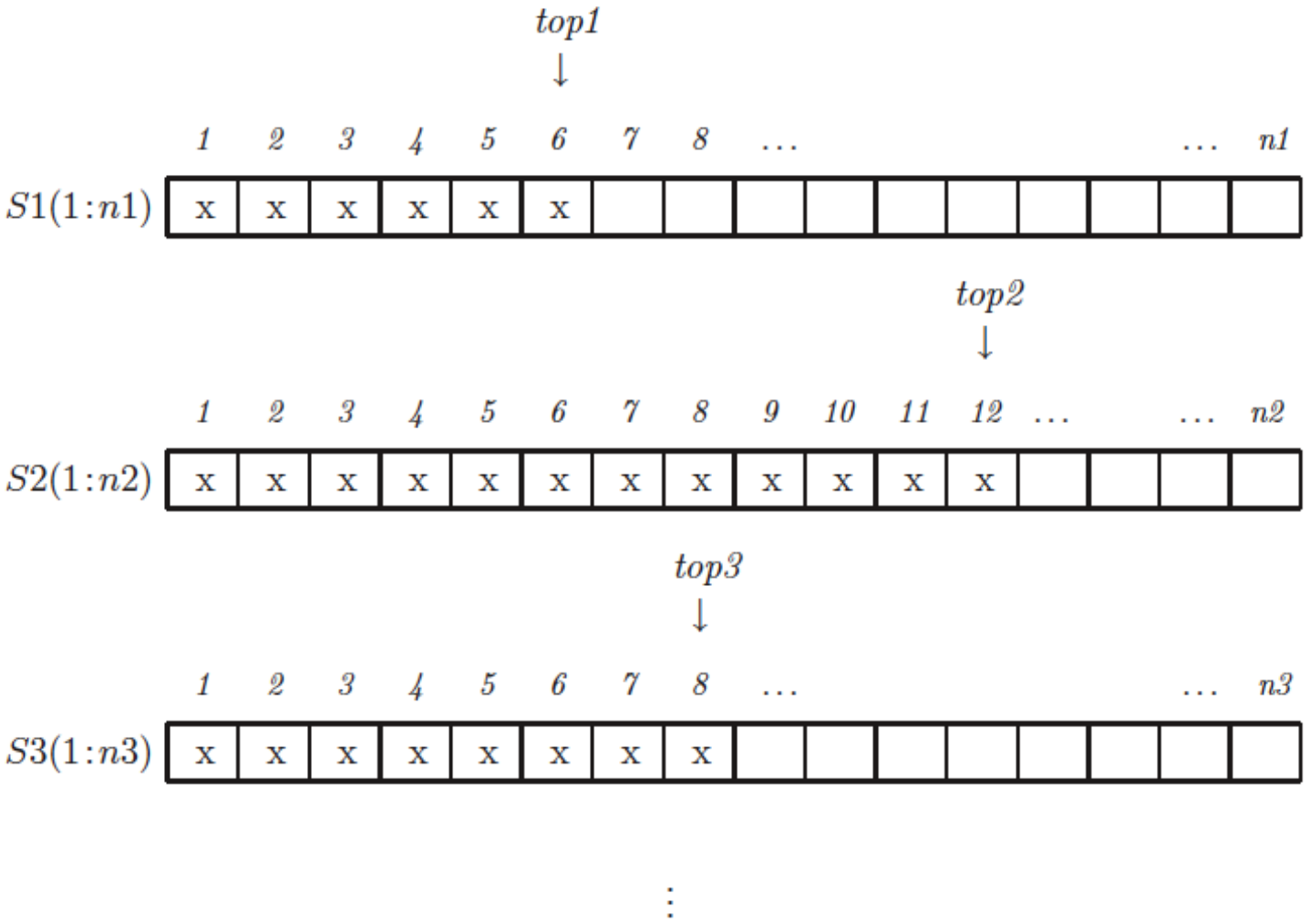
Algorithm in EASY code

```
1     procedure InP(string)
▷ Given an input string 'string' on the alphabet {a, b, c}, right padded with a blank, and
▷ a function NEXTCHAR(string) which returns the next symbol in 'string', procedure
▷ InP returns true if 'string' is in P and false, otherwise.
2     call InitStack(𝕊)
3     char ← NEXTCHAR(string)
4     while char ≠ 'c' do
5         if char = ' ' then return(false)
6                       else [call PUSH(𝕊, char); char ← NEXTCHAR(string)]
7     endwhile
8     while not IsEmptyStack(𝕊) do
9         char ← NEXTCHAR(string)
10        call POP(𝕊, x)
11        if char ≠ x then return(false)
12    endwhile
13    if NEXTCHAR(string) ≠ ' ' then return(false)
14                                else return(true)
15    end InP
```

Outline

- What is a stack?
- Sequential Implementation
- Linked Implementation
- Application: Pattern recognition
- **Multi-stack**

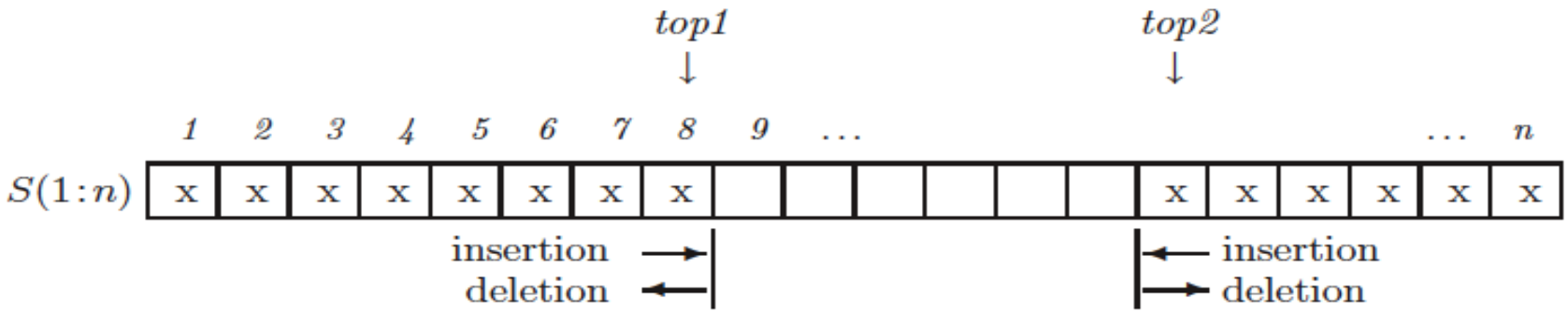
Consider this...



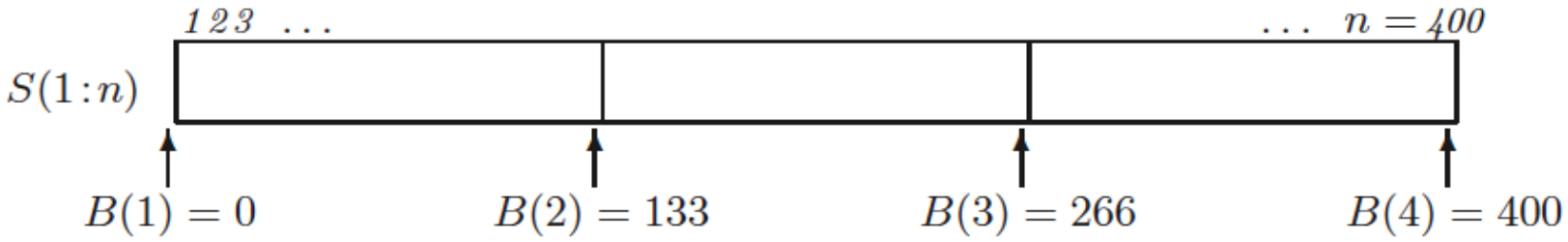
About the previous figure

- One array for each stack
  - All of them are disjoint, i.e., each are maintained in separate non-contiguous addresses in memory
- Space utilization not optimized
  - If one stack overflows, empty space from other stacks *cannot* be used to accommodate item to be inserted
- Re-implement such that one array can have several stacks
  - Not really an uncommon practice
  - Notion of a ***multi-stack***

1 array, 2 stacks



1 array, 3 (or even more) stacks



- Setting boundaries

$$
\begin{aligned}
B(i) &= \lfloor n/m \rfloor (i-1) \qquad 1 \le i \le m \\
B(m+1) &= n
\end{aligned}
$$

1 array, 3 (or even more) stacks

- Push

procedure MPUSH($\mathbb{MS}, i, x$)
if $T(i) = B(i+1)$ **then call** MSTACKOVERFLOW($\mathbb{MS}, i$)
$T(i) \leftarrow T(i) + 1$
$S(T(i)) \leftarrow x$
end MPUSH

- Pop

procedure MPOP($\mathbb{MS}, i, x$)
if $T(i) = B(i)$ **then call** MSTACKUNDERFLOW
else $[x \leftarrow S(T(i)); T(i) \leftarrow T(i) - 1]$
end MPOP

Reallocating free space

- ***Unit-shift technique***
  - ***If stack i overflows, allocate one free space from stack <u>above</u> by <u>shifting elements up</u>***

```
▷ Shift stack elements one cell upward
    for j ← T(k) to T(i) + 1 by −1 do
        S(j + 1) ← S(j)
    endfor
▷ Adjust top and base pointers
    for j ← i + 1 to k do
        T(j) ← T(j) + 1
        B(j) ← B(j) + 1
    endfor
```

Reallocating free space

- ***Unit-shift technique***
  - ***If stacks above stack i are full, we scan stacks <u>below</u> it by <u>shifting elements down</u>***

$\triangleright$ *Shift stack elements one cell downward*
for $j \leftarrow B(k+1)$ **to** $T(i)$ **do**
$S(j-1) \leftarrow S(j)$
endfor
$\triangleright$ *Adjust top and base pointers*
for $j \leftarrow k+1$ **to** i **do**
$T(j) \leftarrow T(j)-1$
$B(j) \leftarrow B(j)-1$
endfor

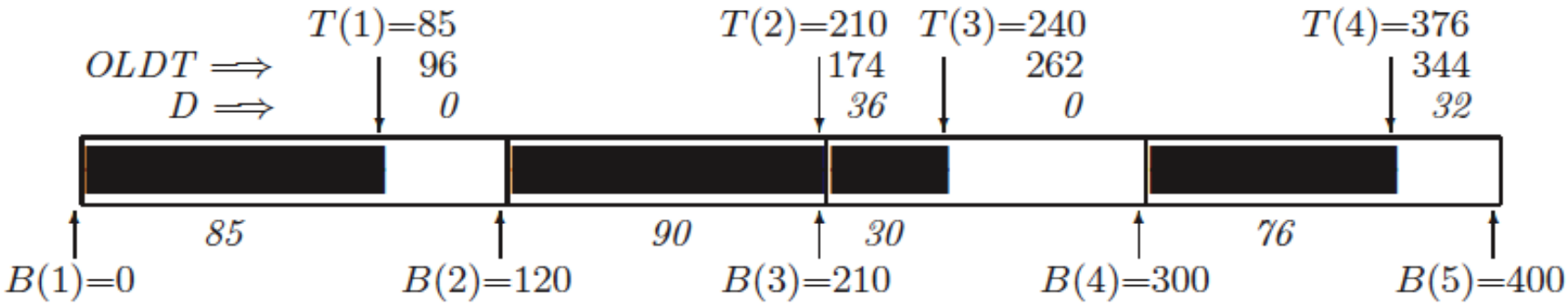
Reallocating free space

- ***Garwick's algorithm***
  - ***Allocate free space to stacks in proportion to their need for space based on some measure***

1. Strip all the stacks of unused cells and consider all of these unused cells as comprising the available or free space.
2. Reallocate one to ten percent of the available space equally among the stacks.
3. Reallocate the remaining available space among the stacks in proportion to need for space as measured by *recent growth*.

Reallocating free space

- ***Garwick- Knuth algorithm***
  - ***10% of free space fixed allocation, 90% distributed based on recent growth***

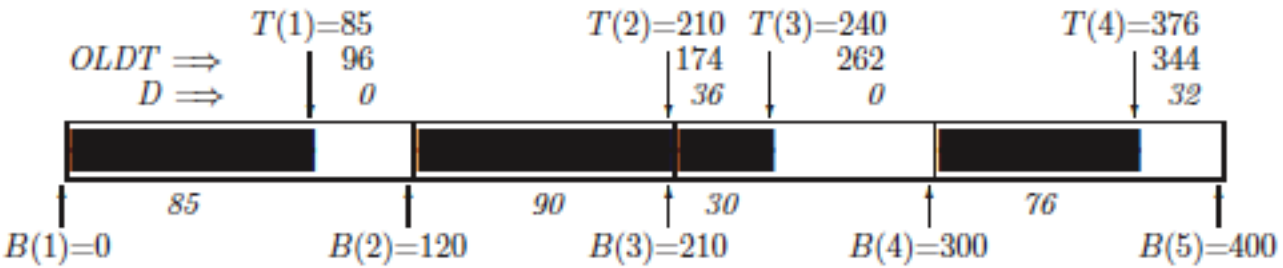


Reallocating free space

- ***Garwick-Knuth algorithm***

1. [*Gather statistics on stack usage.*] The stack sizes $T(j) - B(j)$ and the differences $T(j) - OLDT(j)$ are calculated and indicated Figure 4.13. Note that a negative difference is replaced by zero.

$$freecells = 400 - (85 + 90 + 1 + 30 + 76) = 118$$
$$incr = 36 + 1 + 32 = 69$$



Reallocating free space

- ***Garwick-Knuth algorithm***

2. [*Calculate allocation factors.*]

$$a = (0.10 \times 118)/4 = 2.95$$
$$b = (0.90 \times 118)/69 = 1.54$$

  - ***a represents 10% base allotment***
  - ***b represents allotment per need (from the 90% free space)***

Reallocating free space

- ***Garwick-Knuth algorithm***

3. [*Compute new base addresses.*] Memory reallocation involves essentially determining the new stack boundaries, $NEWB(j), j = 2, 3, \ldots, m$. These new boundaries are found by allocating to each stack the number of cells it is currently using plus its share of the free cells as measured by the parameters a and b. An important aspect of this step is to see to it that the fractional portions are evenly distributed among the stacks. To this end, we define two real variables s and t, as follows:

   s = free space theoretically allocated to stacks $1, 2, \ldots, j-1$
   t = free space theoretically allocated to stacks $1, 2, \ldots, j$

   Then, we calculate the actual number of (whole) free cells allocated to stack j as the difference

   $$\lfloor t \rfloor - \lfloor s \rfloor$$

Reallocating free space

- ***Garwick-Knuth algorithm***

Stack 1: $NEWB(1) = 0; \quad s = 0$

Stack 2: $t = 0 + 2.95 + 0(1.54) = 2.95$
$NEWB(2) = 0 + 85 + \lfloor 2.95 \rfloor - \lfloor 0 \rfloor = 87; \quad s = 2.95$

Stack 3: $t = 2.95 + 2.95 + (36 + 1)(1.54) = 62.88$
$NEWB(3) = 87 + 91 + \lfloor 62.88 \rfloor - \lfloor 2.95 \rfloor = 238; \quad s = 62.88$

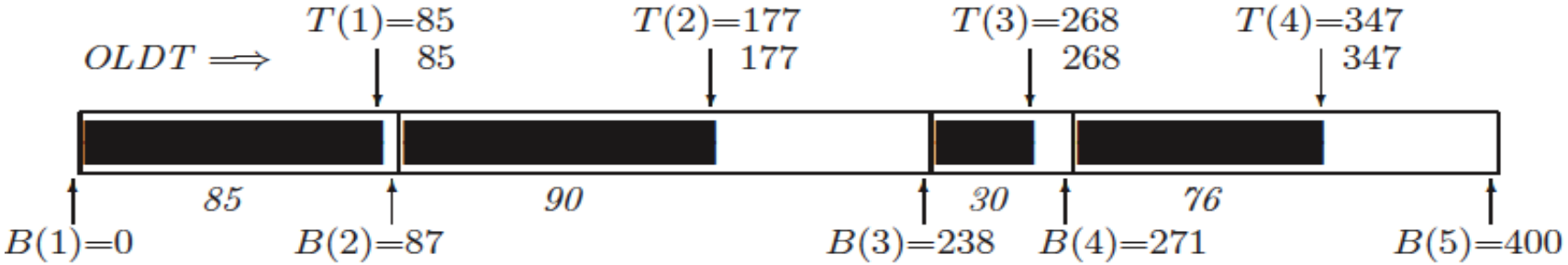
Stack 4: $t = 62.88 + 2.95 + 0(1.54) = 65.83$
$NEWB(4) = 238 + 30 + \lfloor 65.83 \rfloor - \lfloor 62.88 \rfloor = 271$

$t = 65.83 + 2.95 + 32(1.54) = 118.06$
$NEWB(5) = 271 + 76 + \lfloor 118.06 \rfloor - \lfloor 65.83 \rfloor = 400 \qquad$ (O.K.)

Reallocating free space

- ***Garwick-Knuth algorithm***

4. [*Shift stacks to their new boundaries.*] Careful consideration should be given to this step. It is important that no data item is lost (for instance, overwritten) when the stacks are moved. To this end, stacks which are to be shifted down are processed from left to right (i.e., by increasing stack number), while stacks which are to be shifted up are processed from right to left. The same method applies to the individual data items. Data items that are moved down are processed in the order of increasing addresses (indices), and data items that are moved up are processed in the order of decreasing addresses. In this manner, no useful data is overwritten.



Thank you
for your attention.

Questions ...