Overview: Stacks, Queues, Lists

- Sequences of data items appear in many applications;
  → How to represent them in memory?
  → What are typical and specific operations? (define ADTs)
  → How to implement them?

- ADT Stack
  → important applications (recursion, evaluation of expressions)
  → representation in contiguous memory,
  → representation in linked memory

- ADT Queue and its representations

- ADT Array List

- ADT Node List

ADT Stack (Last In First Out)

Operations:

- **Top(S)** returns the top element of stack S or an error code, if S is empty
- **Pop(S)** removes and returns the top element of S or an error code, if S is empty
- **Push(S, x)** adds x on the top of S
- **MakeEmptyStack()** creates a new, empty stack
- **IsEmptyStack(S)** returns true iff S is empty

Typical applications of ADT Stack

- Implementation of recursive procedures,
- Evaluation of arithmetic expressions,
- Checking correctness of parentheses nesting (e.g. XML tags validation)

Computing Factorial

Definition:

\[
\text{fact}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{fact}(n-1) & \text{if } n > 0
\end{cases}
\]

Store intermediate result for each recursion step on a stack S:

\[
\begin{align*}
\text{fact}(3) &= 3 \cdot \text{fact}(2) \\
\text{fact}(2) &= 2 \cdot \text{fact}(1) \\
\text{fact}(1) &= 1 \cdot \text{fact}(0) \\
\text{fact}(0) &= 1
\end{align*}
\]

Remember 3 until fact(2) is computed: Push(S, 3)

Remember 2 until fact(1) is computed: Push(S, 2)

Remember 1 until fact(0) is computed: Push(S, 1)

\[\text{fact}(0) = 1; \quad \text{fact}(3) = 1 \cdot \text{Pop}(S) \cdot \text{Pop}(S) \cdot \text{Pop}(S)\]

⇒ space consumption is \(O(n)\)
Stack application: Implementing recursive procedures

A recursive call is *tail-recursive* if
the first instruction after returning from it is **return**.

- stack not needed: everything on stack will be discarded
- tail-recursive functions can be rewritten using iteration

The recursive call in `fact` is **not** tail-recursive:

```
function fact (integer n) : integer
if n = 0 then return 1
else f ← n · fact(n − 1); return f
```

The first instruction after returning from the recursive call is **multiplication** → `n` must be kept on stack

Tail Recursion

Both recursive calls can be eliminated:

```
function binarySearch (T[a..b], int K) : int
if a > b then return −1
middle ← [(a + b)/2]
if K = T[middle] then
    return middle
else if K < T[middle] then
    { was: return binarySearch(T[a..middle − 1], K) }
    b ← middle − 1
else { K > T[middle] }
    { was: return binarySearch(T[a..middle + 1], K) }
    a ← middle + 1
goto (1)
```

Tail-recursive function

Consider `binarySearch`:

```
function binarySearch (T[a..b], int K) : int
if a > b then return −1
middle ← [(a + b)/2]
if K = T[middle] then
    return middle
else if K < T[middle] then
    return binarySearch(T[a..middle − 1], K)
else { K > T[middle] }
    return binarySearch(T[middle + 1..b], K)
```

Both recursive calls are **tail-recursive**.
Tail-recursive Factorial Function

The function `fact` can be rewritten by introducing a help function:

function \texttt{fact} (integer \( n \)) : integer
return \texttt{fact2}(n, 1)

function \texttt{fact2} (integer \( n \), integer \( f \)) : integer
if \( n = 0 \) then
return \( f \)
else
return \texttt{fact2}(\( n - 1 \), \( n \cdot f \))

\texttt{fact2} is tail-recursive.

⇒ space consumption after recursion elimination will be \( O(1) \)

Implementing a Stack in Contiguous Memory (1)

Implementing a Stack in Linked Memory

### Implementing a Stack in Contiguous Memory (1)

![Stack Diagram](image)

function \texttt{MakeEmptyStack}() : pointer
\( S \leftarrow \text{NewCell}(\text{StackTableHead}) \)
\( A(S) \leftarrow \text{NewCell}(\text{table < dataitem>}[0..n-1]) \)
\( \text{Length}(S) \leftarrow 0 \)
return \( S \)

procedure \texttt{Push}( pointer \( S \), dataitem \( x \))
if \( \text{Length}(S) = n \) then error
else
\( A(S)[\text{Length}(S)] \leftarrow x \)
\( \text{Length}(S) \leftarrow \text{Length}(S) + 1 \)

function \texttt{Pop}( pointer \( S \)) : dataitem
if \( \text{Length}(S) = 0 \) then error
else
return \( A(S)[\text{Length}(S)] \)

### Implementing a Stack in Linked Memory

![Linked Stack Diagram](image)

function \texttt{MakeEmptyStack}() : pointer
\( S \leftarrow \text{NewCell}(\text{StackListHead}) \)
\( \text{ptop}(S) \leftarrow \Lambda \)
return \( S \)

procedure \texttt{Push}( pointer \( S \), dataitem \( x \))
\( y \leftarrow \text{NewCell}(\text{StackListItem}) \)
\( \text{Info}(y) \leftarrow x \)
\( \text{Next}(y) \leftarrow \text{ptop}(S) \)
\( \text{ptop}(S) \leftarrow y \)

+ maximum size need not be known in advance
− call to \( \text{NewCell} \) in each \texttt{Push} operation and to \( \text{FreeCell} \) in each \texttt{Pop} operation

### Implementing a Stack in Contiguous Memory (2)

function \texttt{Top}( pointer \( S \)) : dataitem
if \( \text{Length}(S) = 0 \) then error
else
return \( A(S)[\text{Length}(S) - 1] \)

function \texttt{IsEmptyStack}( pointer \( S \)) : boolean
return \( \text{Length}(S) = 0 \)

All operations take \( O(1) \) time.
Implementing a Queue by a Ring Buffer/Circular Array

ADT Queue (First In First Out)

Operations:

- **Front** \( Q \) returns the first element of \( Q \)
- **Dequeue** \( Q \) removes and returns the first element of \( Q \)
- **Enqueue** \( Q, x \) adds \( x \) at the end of \( Q \)
- **MakeEmptyQueue**() creates a new, empty queue
- **IsEmptyQueue** \( Q \) returns \( true \) iff \( Q \) is empty

Typical application of ADT Queue:

Serving requests in incoming order.

Did you try to book a SAS ticket by phone?
Lists

A list $L$ is a sequence of elements $\langle x_0, \ldots, x_{n-1} \rangle$

- **size** or **length** $|L| = n$
- **empty** list $\langle \rangle$ with length 0
- Selection by index $i$ (sometimes called **rank**): selects the $i$-th element, $x_i$ where $i$ is an integer in the range $0..(n-1)$;
- Selection by actual **position**, e.g. **first** element of $L$ or **last**, **next**, **prev**, ... **Position** abstracts from indexing.

→ ADT Array List: using **rank** [Goodrich, Tamassia 6.1]
→ ADT Node List: using **position** [Goodrich, Tamassia 6.2]

ADT Array List

Domain: lists

Operations on a vector $S$

- **size**() returns $|S|$
- **isEmpty**() returns **true** if $|S| = 0$ and **false** otherwise
- **elemAtRank**$(i)$ returns $S[i]$; error if $i < 0$ or $i > size() - 1$
- **insertAtRank**$(i, x)$ inserts $x$ as a new element at rank $i$: increases the size; error if $i < 0$ or $i > size()$
- **removeAtRank**$(i)$ removes and returns the $i$-th element of $S$: decreases the size; error if $i < 0$ or $i > size() - 1$

ADT Node List (1)

Domain: lists

Operations on a list $L$

- **size**($L$) returns $|L|$
- **isEmpty**() returns **true** if $|S| = 0$ and **false** otherwise
- **first**() returns the position of the first element of $L$; error if $L$ is empty
- **last**() returns the position of the last element of $L$; error if $L$ is empty
- **prev**$(p)$ returns the position of the element of $L$ preceding $p$; error if $p$ is the first position.
- **next**$(p)$ returns the position of the element of $L$ following $p$; error if $p$ is the last position.

...cont. next slide

ADT List (2)

Update operations

- **insertFirst**$(x)$ insert a new element $x$ as the first element of $L$, return the position of $x$.
- **insertLast**$(x)$ insert a new element $x$ as the last element of $L$, return the position of $x$.
- **insertBefore**$(p, x)$ insert a new element $x$ before position $p$ of $L$, return the position of $x$.
- **insertAfter**$(p, x)$ insert a new element $x$ before position $p$ of $L$, return the position of $x$.
- **remove**$(p)$ remove from $L$ and return the element at position $p$ of $L$, return the position of $x$. 
Singly Linked Lists

Implementing ADT Vector, ADT List

- Contiguous memory representation: elements stored in a table/ array (e.g. Dynamic Vectors Lecture 2).
  - `elem` retrieval operations in $O(1)$;
  - what about other operations?
- Singly linked lists: see next slide.
  - `positions` implemented as pointers
  - `prev, insert Before` require list traversal.
- Doubly linked lists: see below.