Basic Abstract Data Types (I)

Hwansoo Han
Basic ADTs

- List
  - General list
  - Stack
  - Queue

- Mapping
  - Associative store
List

- Flexible structure
  - Can grow and shrink on demand
  - Elements can be accessed, inserted, or deleted at any position
  - Two lists can be concatenated
  - A list can be split into sublists
List

Mathematical definition

A sequence of zero or more elements of a given type

\[ a_1, a_2, \ldots, a_n \]

where \( n \geq 0 \),

- \( a_i \) represents a list element,
- each \( a_i \) is of type \( \text{elementtype} \),

The length of the list is \( n \),

- \( a_1 \) is the first element,
- \( a_n \) is the last element
List

- Linearly ordered

\[ a_1, a_2, \ldots, a_n \]

linearly ordered according to their position

- \( a_i \) precedes \( a_{i+1} \)
- \( a_i \) follows \( a_{i-1} \)
- \( a_i \) is at position \( i \)

Special positions: FIRST(L), LAST(L), END(L)
List Operations

Operations on List $L$

- **INSERT** ($x$, $p$, $L$)  // insert element $x$, position $p$
- **LOCATE** ($x$, $L$)  // return position of $x$
- **RETRIEVE** ($p$, $L$)  // return the element at position $p$
- **DELETE** ($p$, $L$)  // eliminate the element at position $p$
- **NEXT** ($p$, $L$)  // return the position following position $p$
- **PREVIOUS** ($p$, $L$)  // return the position preceding position $p$
- **MAKENULL** ($L$)  // make an empty list and return END($L$)
- **FIRST** ($L$)  // return the first position
- **END** ($L$)  // return the next position of last position
- **PRINTLIST** ($L$)  // print all elements in $L$ in order
List Operations

- **Special cases**
  - No matching element
    - LOCATE \( (x, L) \) - returns END\( (L) \)
  - Empty List
    - FIRST \( (L) \) - returns END\( (L) \)

- **Erroneous returns (or undefined behaviors)**
  - Positions in arguments is invalid
    - INSERT \( (x, 100, L) \) when length of \( L \) is 10
  - Unavailable positions
    - PREVIOUS \( (1, L) \), NEXT \( (\text{END}(L), L) \)
Implementation of Lists

- Multiple ways to implement ADT List
  - Array implementation
  - Pointer implementation

Diagram: Array implementation of a list with elements $a_1, a_2, \ldots, a_n$ and a header. The list is empty when the header is not present.
Array Implementation of Lists

- Elements in contiguous cells of an array
  - Easy to traverse the list
  - Easy to insert/delete an element at the tail of the list
  - Difficulty to insert/delete an element in the middle
    - Requires shifting all the following elements
Array Implementation of Lists

\begin{verbatim}
const
    max_length = 100 \{ some suitable constant \};

type
    LIST = record
        elements: array[1..max_length] of elementtype;
        last: integer
    end;

    position = integer;

function END ( var L: LIST ): position; \$
    begin
        return (L.last + 1)
    end; \{ END \}
\end{verbatim}
Array Implementation of Lists

- Implementation of LIST operations
  - INSERT \((x, p, L)\)
  - DELETE \((p, L)\)
  - LOCATE \((x, L)\)
  - MAKENULL \((L)\)
  - NEXT \((p, L)\)
  - PREVIOUS \((p, L)\)
Array Implementation of Lists

- **Validity checks for position** $p$
  - Check if $L$ is full for INSERT
  - Check if $p$ is valid
    - $1 \leq p \leq L.last + 1$ for INSERT
    - $1 \leq p \leq L.last$ for DELETE, LOCATE

- **Update** $last$
  - Increase by 1 for INSERT
  - Decrease by 1 for DELETE
Pointer Implementation of Lists

- Use singly-linked cells, AKA *singly linked list*
  - Use pointers to link successive list elements
  - No need to maintain contiguous memory for list elements – easy to grow or shrink without shifting
  - Require extra space for pointers

![Diagram of singly linked list](image)
type

celltype = record
    element: elementtype;
    next: ↑ celltype
end;
LIST = ↑ celltype;
position = ↑ celltype;
function END ( L: LIST ): position;
    { END returns a pointer to the last cell of L }
var
    q: position;
begin
    q := L;
    while q↑.next <> nil do
        q := q↑.next;
    return (q)
end; { END }
Pointer Implementation of Lists

- Implementation of LIST operations
  - INSERT \( (x, p, L) \)
  - DELETE \( (p, L) \)
  - LOCATE \( (x, L) \)
  - MAKENULL \( (L) \)
  - NEXT \( (p, L) \)
  - PREVIOUS \( (p, L) \)
Insertion of a new element $x$ into a list represented by pointers $p$, $a$, and $b$.

1. Copy $a$'s value to $temp$.
2. Link $temp$ to $a$.
3. Adjust pointers to link $x$ to $a$ and $b$ to $temp$.
4. Update pointers to link $a$ to $x$. 

Hence, $x$ is inserted into the list.
Pointer Implementation of Lists

- DELETE \((p, L)\)
Pointer Implementation of Lists

- Meaning of *position* in singly linked list
  - The next cell of the cell pointed by position $p$

- LOCATE ($x, L$)
  - Returns which?

1. $p \rightarrow a \rightarrow b$
2. $p \rightarrow q \rightarrow x \rightarrow b$
3. $p \rightarrow r \rightarrow x \rightarrow b$
Pointer Implementation of Lists

- **PREVIOUS** ($p$)
  - Need to traverse from the first element
  - Time consuming operation if list is very long

- Doubly linked list
  - Traverse both forwards and backwards efficiently
  - No need to change the meaning of $position$
Pointer Implementation of Lists

- DELETE \((p, L)\) for doubly linked list
  - Meaning of position \(p\) is the same as array implementation