
List ADT:

Definition:

*   A set of objects stored in a 2-way chained contiguous linear sequence

Operations:

*   listCreate
        - Create new list initialized to empty
    PRE:
        void

    POST:
        new list is allocated and initialized; returns a
        reference to new list

    ERRORS:
        memory allocation failure (allocation)

*   listDelete
        - De-allocate all memory associated with provided
          list object

    PRE:
        void

    POST:
        this list is destroyed; returns number of elements
        that were in the list

    ERRORS:
        void

*   listClear
        - De-allocate all memory associated with element
          structures of provided list, resetting state of
          list to empty

    PRE:
        void

    POST:
        this list is empty; returns number of elements that
        were in the list

    ERRORS:
        void

*   listHead
        - Adjust position to the beginning of provided list
        - NOTE: this operation is non-destructive on an
          empty list

    PRE:
        list is non-empty

    POST:
        current & position reference the first element in
        this list; returns a reference to this

    ERRORS:
        list is empty

*   listTail
        - Adjust position to the end of provided list
        - NOTE: this operation is non-destructive on an
          empty list

    PRE:
        list is non-empty

    POST:
        current & position reference the last element in
        this list; returns a reference to this

    ERRORS:
        list is empty

*   listPrev
        - Move the the previous element in the provided list

    PRE:
        list is non-empty; list is not positioned at the
        first element

    POST:
        current & postition now reference the element
        preceeding the current one in this list; returns a
        reference to this

    ERRORS:
        list is empty
        already at beginning of list

*   listNext
        - Move the the next element in the provided list

    PRE:
        list is non-empty; list is not positioned at the
        last element

    POST:
        current & position now reference the element
        succeeding the current one in this list; returns
        a reference to this

    ERRORS:
        list is empty
        already at end of list

Stack ADT:

Definition:

*   Last-In/First-Out sequence where additions and deletions occur at the top.

Operations:

*   stackCreate
        - Create new stack initialized to empty
    PRE:
        void

    POST:
        new stack is allocated and initialized; returns a

```
        reference to new stack

    ERRORS:
        memory allocation failure (allocation)
*   stackDelete
        - De-allocate all memory associated with provided
          stack object

    PRE:
        void

    POST:
        this stack is destroyed; returns number of elements
        that were in the stack

    ERRORS:
        void
*   stackClear
        - De-allocate all memory associated with element
          structures of provided stack, resetting state of
          stack to empty

    PRE:
        void

    POST:
        this stack is empty; returns number of elements that
        were in the stack

    ERRORS:
        void
*   stackIsEmpty
        - Determine if provided stack is empty

    PRE:
        void

    POST:
        returns whether or not this is empty

    ERRORS:
        void
*   stackLength
        - Determine number of elements (length) in
          provided stack

    PRE:
        void

    POST:
        returns length

    ERRORS:
        void
*   stackPop
        - Removes an item from the stack and returns its value.

    PRE:
        stack is non-empty

    POST:
        returns a reference to the data removed from the top of the stack

    ERRORS:
        stack is empty
*   stackPush
        - Adds an item to the top of the stack
```

```
    PRE:
        void

    POST:
        a new data reference is inserted into this stack at the top

    ERRORS:
        void

-----------------------------------------------------------------------

Queue ADT:

-----------------------------------------------------------------------

Definition:

*   First-In/First-Out sequence where additions occur at the end, and deletions occur at the

Operations:

*   queueCreate
        - Create new queue initialized to empty
    PRE:
        void

    POST:
        new queue is allocated and initialized; returns a
        reference to new queue

    ERRORS:
        memory allocation failure (allocation)
*   queueDelete
        - De-allocate all memory associated with provided
          queue object

    PRE:
        void

    POST:
        this queue is destroyed; returns number of elements
        that were in the queue

    ERRORS:
        void
*   queueClear
        - De-allocate all memory associated with element
          structures of provided queue, resetting state of
          queue to empty

    PRE:
        void

    POST:
        this stack is empty; returns number of elements that
        were in the stack

    ERRORS:
        void
*   queueIsEmpty
        - Determine if provided queue is empty

    PRE:
        void

    POST:
```

```
        returns whether or not this is empty

    ERRORS:
        void
*   queueLength
        - Determine number of elements (length) in
          provided queue

    PRE:
        void

    POST:
        returns length

    ERRORS:
        void
*   queueDepart
        - Removes an item from the front of the queue and returns its value.

    PRE:
        queue is non-empty

    POST:
        returns a reference to the data removed from the first in the queue.

    ERRORS:
        stack is empty
*   queueArrive
        - Adds an item to the end of the queue

    PRE:
        void

    POST:
        a new data reference is inserted at the last in the queue

    ERRORS:
        void

-----------------------------------------------------------------------

Advantages and Disadvantages of Contiguous Memory

-----------------------------------------------------------------------

PROS:
* Fast access times
* Very simple to move to the Nth item in the list.
* Easy to code

CONS:
* Slow insertion and deletion operations because you need to slide around the contained ele

-----------------------------------------------------------------------

Parentheses Checker

-----------------------------------------------------------------------

To use the parentheses checker, type ./Paren <filename>

Here is some sample input/output for your benefit!

INPUT FILE parentest1:
5 * (6 + 53)

OUTPUT DISPLAY:
```

```
Parsed successfully.

INPUT FILE parentest2:
((((3 / 9) - 2)

OUTPUT DISPLAY:
((((3 / 9) - 2)
 ^
No match for opening ( on line 1, column 2
((((3 / 9) - 2)
^
No match for opening ( on line 1, column 1
Parse failed with 2 matching error(s).

INPUT FILE parentest3:
(a
    - b)(
)
        ())

OUTPUT DISPLAY:
                ())
                  ^
Unexpected closing ) on line 4, column 19
Parse failed with 1 matching error(s).
```

Testing methodology

To test my ADTs, I utilized ListMenu.c, StackMenu.c, and QueueMenu.c for my List, Stack, and Queue respectively. I ran through each of the operations by simply messing around with each of the menu items, and repeating the ones that would be playing a lot with the memory.