Overview

- Stacks
- Queues
- Adding to the Tail of a List
- Efficiency Issues
- Queue Structure
- Stack Application: Postfix Calculator

Stacks and Queues

- Stacks and Queues are examples of Abstract Data Types
- Stacks and Queues are used in many computing applications, as well as forming auxiliary data structures for common algorithms, and appearing as components of larger structures.

Stacks

- A stack is a collection of items such that the last item to enter is the first one to exit, i.e. “last in, first out” (LIFO)
- Based on the idea of a stack of books, or plates

Stack Functions

- Essential Stack functions:
  - push() // add new item to stack
  - pop() // remove top item from stack
- Additional Stack functions:
  - top() // fetch top item (but don’t remove it)
  - size() // number of items
  - isEmpty()
Stack Applications

- page-visited history in a Web browser
- undo sequence in a text editor
- checking for balanced brackets
- HTML tag matching
- postfix calculator
- chain of function calls in a program

Queues

- a queue is a collection of items such that the first item to enter is the first one to exit, i.e. “first in, first out” (FIFO)
- based on the idea of queueing at a bank, shop, etc.

Queue Functions

- Essential Queue functions:
  - enqueue() // add new item to queue
  - dequeue() // remove front item from queue
- Additional Queue functions:
  - front() // fetch front item (but don’t remove it)
  - size() // number of items
  - isEmpty()

Queue Applications

- waiting lists, bureaucracy
- access to shared resources (printers, etc.)
- phone call centres
- multiple processes in a computer
Implementing Stacks and Queues

- A stack can be implemented using a linked list, by adding and removing at the head [push() and pop()]
- For a queue, we need to either add or remove at the tail
  - Can either of these be done efficiently?

Adding to the Tail of a List

- Adding an item at the tail is achieved by making the last node of the list point to the new node
- We first need to scan along the list to find the last item

Adding to the Tail of a List

```
Lnode *add_to_tail(Lnode *new_node, Lnode *head)
{
    if (head == NULL) {  // list is empty
        head = new_node;
    } else {
        Lnode *node = head;
        while (node->next != NULL) {
            node = node->next;  // scan to end
        }
        node->next = new_node;
    }
    return head;
}
```

Efficiency Issues

Unfortunately, this implementation is very slow. Every time a new item is inserted, we need to traverse the entire list (which could be very large).

We can do the job much more efficiently if we retain a direct link to the last item or “tail” of the list:

```
if (tail == NULL) {  // list is empty
    head = node;
} else {
    tail->next = node;
}
```

Note: There is no way to efficiently remove items from the tail. (Why?)
Queue Structure

We can use this structure to implement a queue efficiently:

typedef struct queue Queue;

struct queue {
    Lnode *head;
    Lnode *tail;
    int size;
};
Example: queue.c

```c
int main(void) {
    Queue *q = makeQueue();
    Lnode *node;
    int ch;

    while (((ch = getchar()) != EOF) {
        if (ch == '-') {
            node = dequeue(q);
            if (node != NULL)
                printf("Dequeueing \%c\n", node->data);
            free(node);
        } else if (ch == '\n') {
            printList(q->head);
        } else {
            enqueue(makeNode(ch), q);
        }
    }
}
```

Reverse Polish Notation

Some early calculators and programming languages used a convention known as Reverse Polish Notation (RPN) where the operator comes after the two operands rather than between them:

1 2 +
result = 3
3 2 *
result = 6
4 3 + 6 *
result = 42
1 2 3 4 + * +
result = 15

Postfix Calculator

A calculator using RPN is called a Postfix Calculator; it can be implemented using a stack:

- when a number is entered: push it onto the stack
- when an operator is entered: pop the top two items from the stack, apply the operator to them, and push the result back onto the stack.
int main(void) {
    Lnode *list = NULL;
    int num;
    int a, b, num;
    while ((ch = getc(stdin)) != EOF) {
        if (ch == '
') {
            printf("Result: %d\n", list->data);
        }
        else if (isdigit(ch)) {
            ungetc(ch, stdin); // put first digit back
            scanf("%d", &num); // now scan entire number
            list = push(makeNode(num), list);
        }
        else if (ch == '+' || ch == '-' || ch == '*') {
            else if (list != NULL) {
                a = list->data; // fetch top item
                list = pop(list);
                if (list != NULL) {
                    b = list->data; // fetch 2nd item
                    list = pop(list);
                    switch (ch) {
                        case '+': num = b + a; break;
                        case '-': num = b - a; break;
                        case '*': num = b * a; break;
                    }
                    list = push(makeNode(num), list);
                }
            }
        }
    }
}