Restricted Linked List

Linear List:
- Ordered sequence of nodes a_1, a_2, … a_n where the number of nodes varies

Restricted Linear List:
- A Linked List with restrictions placed on where/how we can add/remove items from list

Stack
- Add, remove from end or top
- LIFO – Last In First Out
- Operations: Push, Pop, Top, initialize, empty, full
- Conditions: underflow – tired to pop and empty stack; overflow – tried to push onto full stack

Used extensively in Computer Science:
- Backtracking (keep track of postponed obligations) with stack
  - "walking" a tree
  - Solving optimization problems (e.g. shortest path)
  - Nested/recursive function calls
- Compilers
  - Parsing, recognizing Context Free Languages
  - Translations, infix to postfix
  - Expression evaluation, balanced parentheses check
  - Eliminate recursion

e.g. check string s for balanced parentheses:

```
for i from 0 to (strlen(s) - 1)
  if S[i] == '(' || S[i] == '[' || s[i]== '{'
    push(S[i], stk)
  else if S[i] == ')' || S[i] == ']' || s[i]== '}'
    if empty(stk)
      print "more right parentheses than left"
      return
    else
      pop(stk, item)
      if (S[i] != item))
        print "mismatched parentheses"
        return
  if empty(stk)
    print "balanced"
  else
    print "more left parentheses than right"
```

(	(	B	[	3	]	+	2	)	*	k	)	\0

(	(	B	[	3	[	+	2	)	*	k	)	\0

e.g. Expression Evaluation:

- evaluate postfix expressions

infix: (5+3) * 7	postfix: 5 3+7*
8 * 7	8 7*
56	56

easy to evaluate postfix because no parentheses, can use stacks

1) Add terminator to Strings, e.g. “#”
2) Algorithm:

```
for each S[i], in order
  if S[i] is operand
    push(S[i], stk)
  if S[i] is operator
    pop(stk, A)
    pop(stk, B)
    result = A S[i] B
    push(result, stk)
  if S[i] is terminator
    pop result
```

5	3	+	7	*	#

Since postfix evaluation is so easy, compilers tend to translate infix to postfix THEN evaluate postfix. This is easier than evaluating infix directly

Translating infix to postfix

- Stack
- Precedence/order of operation matrix that specifies action to be taken, depending on:
  1) Next toke in input stream
  2) Token at top of stack

Algorithm:

```
int i = 0, k=-1;

for each item in exp[i]
  if (isOperand(exp[i])) //if the item is an operand, add it to output
    output[++k] = exp[i]
  else if (exp[i] == '(') //if item is (, push it to the stack
    push(stack, exp[i])
  else if (exp[i] == ')')
    // if item is ), pop and output from stack until ( is encountered
    while (!isEmpty(stack) && peek(stack) != '(')
      output[++k] = pop(stack);
      pop(stack)
   else
     while (!isEmpty(stack) && Prec(exp[i]) <= Prec(peek(stack)))
       exp[++k] = pop(stack)
       push(stack, exp[i])

    while (!isEmpty(stack))  // pop all the operators from the stack
        exp[++k] = pop(stack )
```

Order of Operation

1) Parentheses ( )
2) Exponents ^
3) Multiplications and Division * /
4) Addition and Subtraction + –

Examples:

Infix: A * B – C
Postfix: AB*C–

Infix: A * (B – C)
Postfix: ABC– *

Infix: A + B * C – D * E
Postfix: ABC*+DE*–

Infix: ((A + B) * C – D) * E
Postfix: AB+C*D–E*

Sequential Stack implementation C:

```
typedef struct {
  int count;
  itemType items[MAX];
} Stack;

void init (Stack *S) {
  S->count = 0;
}

void push (ItemType x, Stack *S) {
  if S->count == MAX
    overflow();
  else
    S->Items[S->count] = x;
    ++(S-.count);
}
```

`init:`

count	0				
Items	…	…	…	…	…

`push(3):`

count	1				
Items	3	…	…	…	…

`push(4):`

count	2				
Items	3	4	…	…	…

`pop:`

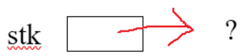
count	1				
Items	3	4	…	…	…

Linked Stack implementation C:

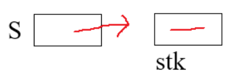
```
typedef struct StackNodeTag{
  itemType Item;
  struct StagNodeTag *link;
} StackNode;

typedef struct {
  StackNode *ItemList;
}Stack;

void init (Stack *S) {
  S->ItemList = NULL;
}
```

```
void push (ItemType x, Stack *S) {
  StackNode *Temp;

   if(Temp=(StackNode*)malloc(sizeof(S
   tackNode)) == NULL)
    overflow();
  else
    Temp->item = x;
    Temp->link = S->ItemList
    S->ItemList = Temp
}
```

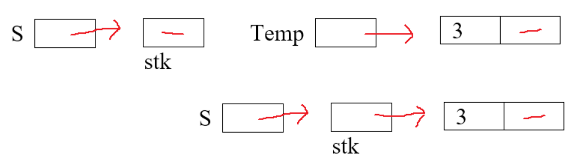
declare



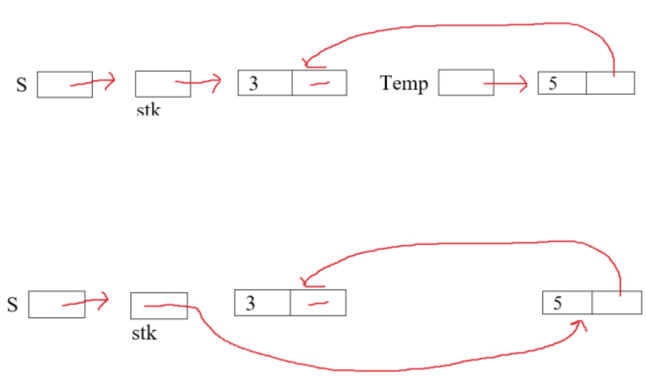
init (&stk)



push (3)



push(5)



Stack implementation in Python:

```
class Stack:
     def __init__(self):
         self.items = []

     def isEmpty(self):
         return self.items == []

     def push(self, item):
         self.items.append(item)

     def pop(self):
         return self.items.pop()

     def peek(self):
         return self.items[len(self.items)-1]

     def size(self):
         return len(self.items)
```

Queue

- Add to end, remove from front (lining up for bus, Timmys)
- FIFO – First In First Out
- Operations: insert, remove, firstItem, init, empty, full
- Conditions: underflow – tried to remove and empty queue; overflow – tried to add onto full queue

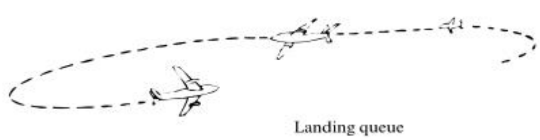
Used extensively in Computer Science:

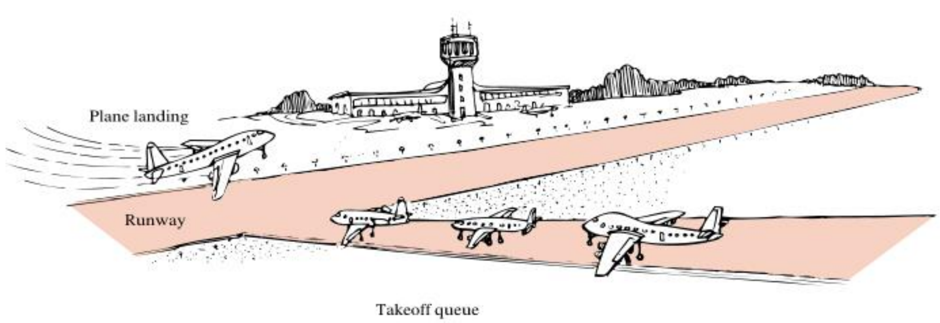
- client/server
  - requests to a website get processed by webserver FIFO
- Operating systems
  - print spools (how much data, printer can handle at one time)
  - print buffer (store jobs to be printed)
  - disk read/write buffer & memory read/write buffer
- Simulation
  - air traffic control system sims
  - traffic flow in Toronto
  - lead testing for systems (webservers, servlet engines, telephone switching systems etc.)

Application of Queues: Simulation of an Airport

Simulation is the use of one system to imitate the behaviour of another system. A computer simulation is a program to imitate the behaviour of the system under study.

1. The same run way is used for both landings and takeoffs
2. One plane can land or take off in a unit of time, but not both
3. A random number of planes arrive in each time unit
4. A plane waiting to land goes before one waiting to takeoff
5. The planes that are waiting are kept in queues landing and takeoff, both of which have a strictly limited size





Sequential Queue Implementation C:

Sequential queues are "circular" array which looks any normal array in memory but accessed as "circular", why?

Visit: https://www.cs.usfca.edu/~galles/visualization/QueueArray.html to see visualization

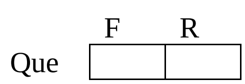
```
typedef  struct {
  int       Count;                  /* number of queue items */
  int       Front;
  int       Rear;
  ItemType  Items[MAXQUEUESIZE];
}Queue;

void InitializeQueue(Queue *Q) {
  Q->Count = 0;              /* Count == number of items in the queue */
  Q->Front = 0;           /* Front == location of item to remove next */
  Q->Rear = 0;                   /* Rear == place to insert next item */
}

void Insert(ItemType R, Queue *Q) {
  if (Q->Count == MAXQUEUESIZE) {
    SystemError("attempt to insert item into a full Queue");
  }
  else {
    Q->Items[Q->Rear] = R;
    Q->Rear = (Q->Rear + 1) % MAXQUEUESIZE;
    ++(Q->Count);
  }
}
```

Linked Queue Implementation C:

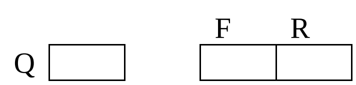
```
typedef  struct  QueueNodeTag {
  ItemType              Item;
  struct  QueueNodeTag  *Link;
}QueueNode;

typedef  struct {
  QueueNode  *Front;
  QueueNode  *Rear;
}Queue;



void InitializeQueue(Queue *Q) {
  Q->Front = NULL;
  Q->Rear  = NULL;
}
```

```
void Insert(ItemType R, Queue *Q) {
  QueueNode *Temp;
  /* attempt to allocate a new
   node*/
  Temp = (QueueNode *)
   malloc(sizeof(QueueNode));

  if (Temp == NULL)
    SystemError("out of memory");
  }
  else {
    Temp->Item = R;
    Temp->Link = NULL;
    if ( Q->Rear == NULL ) {
      Q->Front = Temp;
      Q->Rear = Temp;
    }
    else {
      Q->Rear->Link = Temp;
      Q->Rear = Temp;
    }
  }
}
```

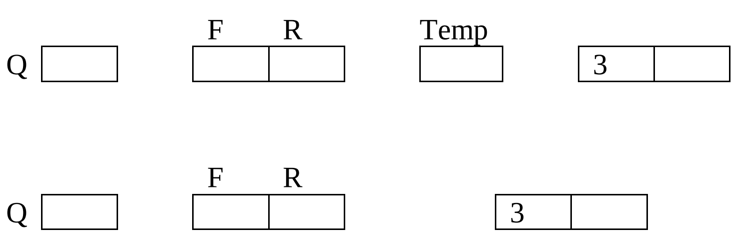
`Queue Que:`



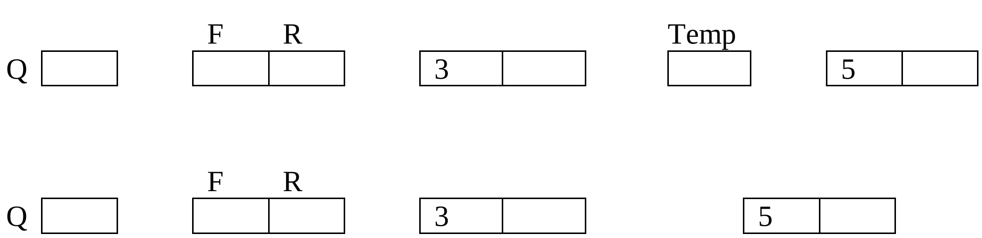
`init (&Que):`



`insert (3, &Que):`



`insert (5, &Que):`



Queue implementation in Python:

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```