BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Data Structures and Algorithms Design

DSECLZG519

Parthasarathy

DSECLZG519 – Contact Session #3
Linear Data Structures

Agenda for CS #3

1) Recap of CS#2
2) Simplified Master's Theorem
3) Linear Data structures
4) Stacks
   - Stack implementation using array
   - Stack applications
     - Expression Evaluation
     - Infix to postfix conversion [Self-Study] – Available in appendix of this presentation
5) Queues
   - Queue implementation using array
   - Circular Queue
   - Applications of Queue
6) Linked List
   - Singly Linked list
   - Doubly Linked list – Insertion & Deletion
7) Stack & Queue implementation using linked list
8) A word on Amortized analysis
9) Q&A

Recap of CS#2

- General Rules and Problems
- Mathematical analysis of Non-recursive algorithms
- Mathematical analysis of Recursive algorithms
  - Iteration vs Recursion
  - Recurrence Tree
  - Iterative Substitution
- Masters Theorem
  - Masters method with an example for each case

Operations on Data

- Typical operations on data:
  - Add data to a data collection (insert)
  - Remove data from a data collection (delete)
  - Ask questions about the data in a data collection (search, exists, update etc.)

Linear Data Structures

- Data is organized in a linear fashion.
- Simple ADTs, example :
  - *Stack*
  - *Queue*
  - Vector
  - Lists
  - Sequences

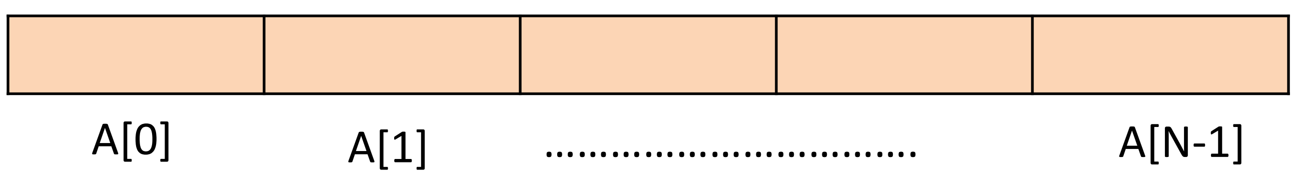
  All these are called *Linear Data Structures*

Non-linear Data Structures

- Data is not organized in a linear fashion.
- Relationship exists among data.
- Examples :
  - Trees
  - Graphs

Arrays

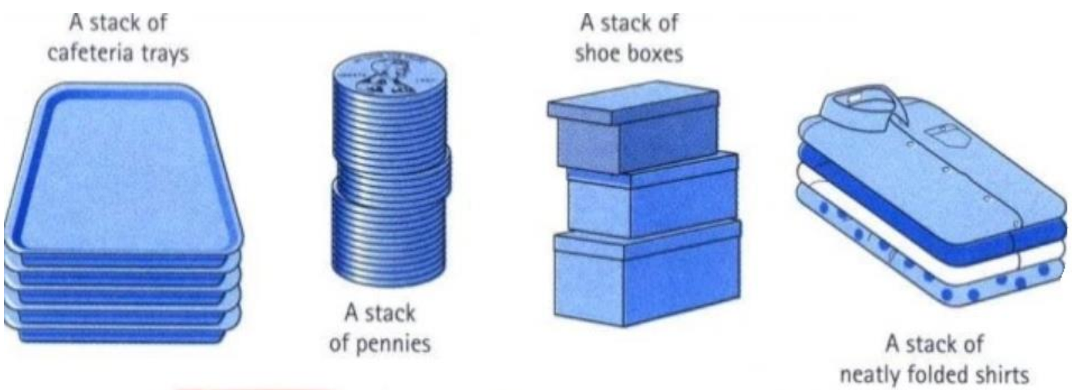
- Array is a finite set of homogeneous elements stored in adjacent/contiguous memory locations. It is represented by a single name.
- Each location element is identified by a single name with its index. The first element is A[0], the second element is A[1] and so on ..
- If there are N elements in the array, the last element will be A[N-1].

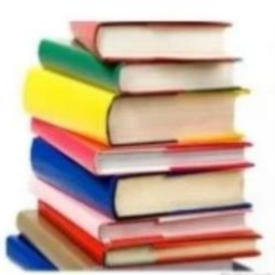


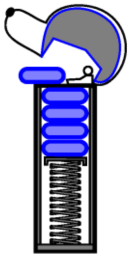
Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (**LIFO**) principle.
- Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Inserting an item is known as "**pushing**" onto the stack. "**Popping**" off the stack is synonymous with removing an item.
- Pushing and popping happens only in one end called the top.

Stacks


A stack of
cafeteria trays
A stack
of pennies
A stack of
shoe boxes
A stack of
neatly folded shirts





Stacks: An Array Implementation

- Create a stack S using an array by specifying a maximum size *N* for our stack.
- The stack consists of an N-element array *S* and an integer variable *t*, the index of the top element in array S.

- Array indices start at 0, so we initialize *t* to -1

Stacks: An Array Implementation

Pseudo code

```
Algorithm size()
return t+1
```

10	20	33	41		
0	1	2	3	4	5

```
Algorithm isEmpty()
 if t == -1

    return true

return false
```

t=-1

0	1	2	3	4	5

```
Algorithm top()
if isEmpty() then
   return Error
return S[t]
```

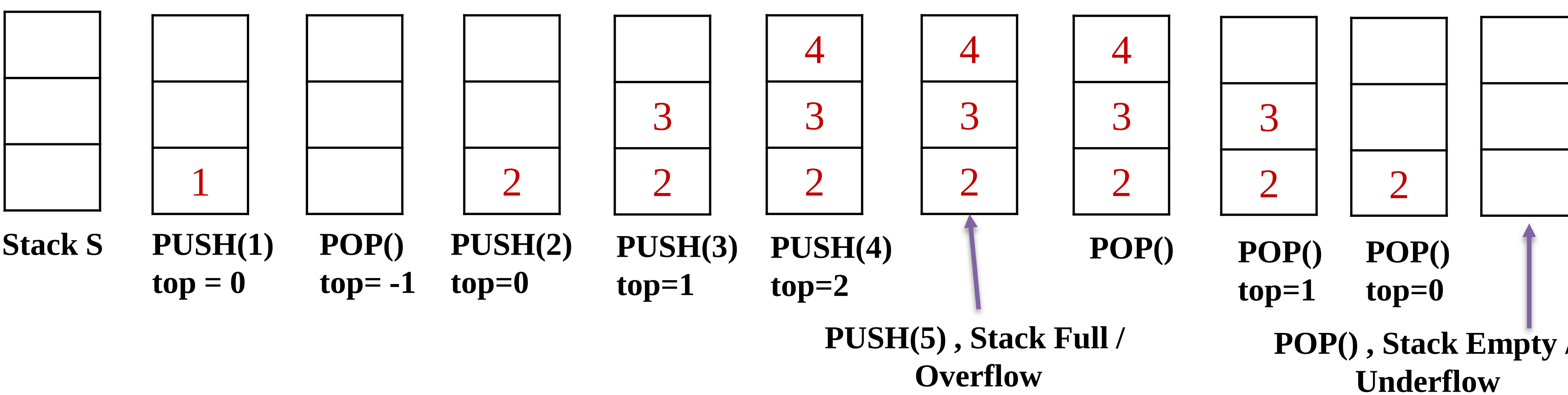
```
Algorithm push(o)
if size()==N then
   return Error/Overflow
t=t+1
S[t]=o
```

```
Algorithm pop()
 if isEmpty() then
   return Error/Underflow
e= S[t]

S[t] = null

t=t-1

return e
```

Operations on Stack Demo

Consider array based implementation of stack S. Assume that the maximum size of the stack is 3 elements. Show the contents of the STACK (trace through) at each step, for the following sequence of operations. Mention exceptions like empty/full if any.

Operations: PUSH (1), POP, PUSH (2), PUSH (3), PUSH (4), PUSH (5), POP, POP, POP, POP



Stacks: An Array Implementation

The array implementation is simple and efficient (methods performed in O(1)).

Disadvantage

- There is an upper bound, *N*, on the size of the stack.
- The arbitrary value *N* may be too small for a given application OR a waste of memory.

Applications of Stack

- Stacks can be used for expression evaluation. *[In class]*
- Stacks can be used for Conversion from one form of expression to another. *[In Appendix of this presentation]*
- Stacks can be used to check parenthesis matching in an expression. *[Self-study/Webinar]*
- Stack data structures are used in backtracking problems (recursion). *[Explore!]*

Forms of Expression

Arithmetic expressions can be written in 3 different forms or notations :

- Infix Notation : Notation in which the operator symbol is placed in between its operands. Ex : A + B , C – D , A * D etc.
- Pre-fix Notation : It is also called as Polish notation and refers to the notation in which the operator symbol is placed **before** its operands *(when operator precedes the operands)*. Ex : +AB, -CD, *AD etc.
- Post-fix Notation : It is also called as reverse polish notation and refers to the notation in which the operator symbol is placed **after** its operands *(when operator follows the operands)*. Ex : AB+ , CD- , AD*

Algorithm: Evaluation of Postfix expression using Stack

Algorithm:

1) Add a right parentheses “)” at the end of the arithmetic expression F
2) Scan F from left to right and repeat step 3 and step 4 for each element of F until the sentinel “)” is encountered.
3) If an operand is encountered, put it onto stack
4) If an operator X is encountered, then :
   a. Remove the 2 top elements from the stack, where n1 is the top element and n2 is the next-to-top element.
   b. Evaluate n2 X n1
   c. Place the result of (b) back on stack.
   Endif
   End of step 2 loop
5) Set value equal to the top element of stack.
6) Exit

X : is any operator

Example: Evaluation of Postfix expression using stack

Evaluate AB+C-BA+C^- where A=1,B=2 and C=3

➔ ***1 2 + 3 – 2 1 + 3 ^ -)***

Symbol Encountered	N1 (top)	N2(2nd top)	Value = N2 op N1	STACK
1				1
2				1 2
+	2	1	1+2 = 3	3
3				3 3
-	3	3	3 -3 = 0	0
2				0 2
1				0 2 1
+	1	2	2+1 = 3	0 3
3				0 3 3
^	3	3	3^3 = 27	0 27
-	27	0	0 – 27 = -27	-27
)	-	-	**-27**	Empty

Hence, the evaluation of this expression leads to the answer **-27**

Exercises

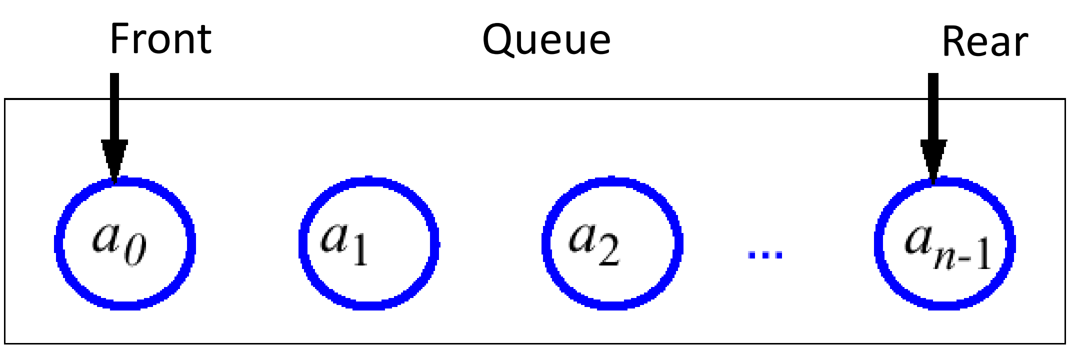
Evaluate the following postfix expressions :

- ABC+*CBA-+* where A=1,B=2 and C=3
- 5 7 5 - * 12 4 * 24 / 6 + +

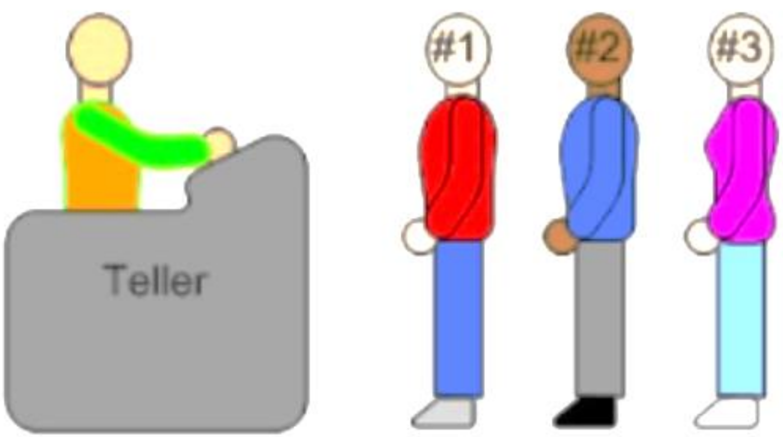
After you solve it, verify your answer by putting the postfix expression in this tool https://www.free-online-calculator-use.com/postfix-evaluator.html

Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues



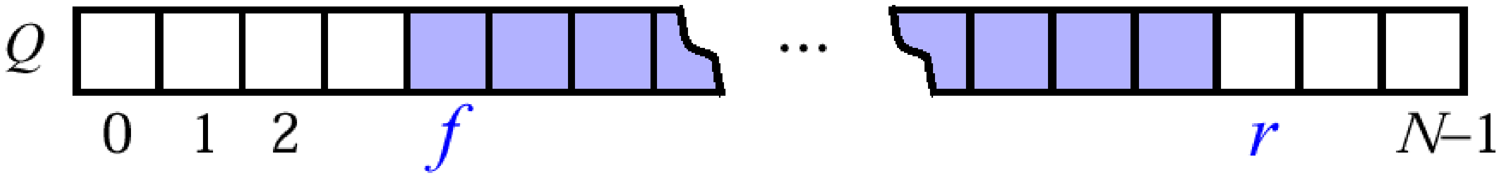
Queues

The **queue** supports three fundamental methods:

- **Enqueue(S:***ADT*, **o:***element***):** - Inserts object *o* at the rear of the queue; an error occurs if the queue is full
- **Dequeue(S:***ADT***):** - Removes the object from the front of the queue; an error occurs if the queue is empty
- **Front(S:***ADT***):***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues: An Array Implementation

- Create a queue using an array
- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, index of the front element (which is the candidate to be removed by a dequeue operation)
  - *r*, index of the next available array cell (cell which can be used in enqueue operation)
  - Initially, f=r=0 and the queue is empty if f=r



Queues

Disadvantage

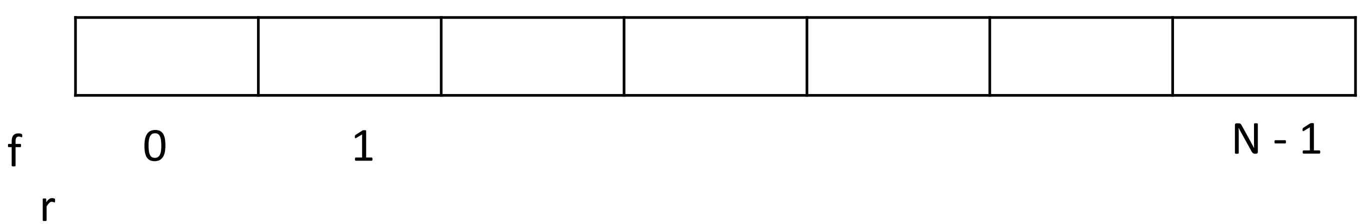
Repeatedly enqueue and dequeue a single element N times.

Finally, f=r=N.

- No more elements can be added to the queue, *though there is space in the queue !!*

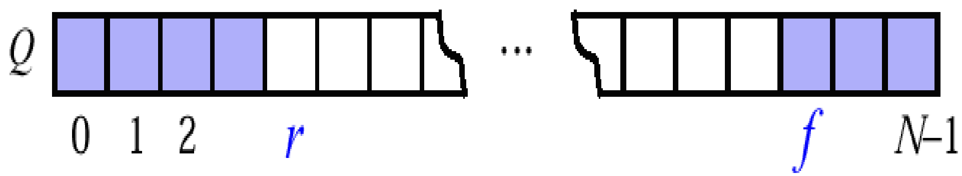
Solution

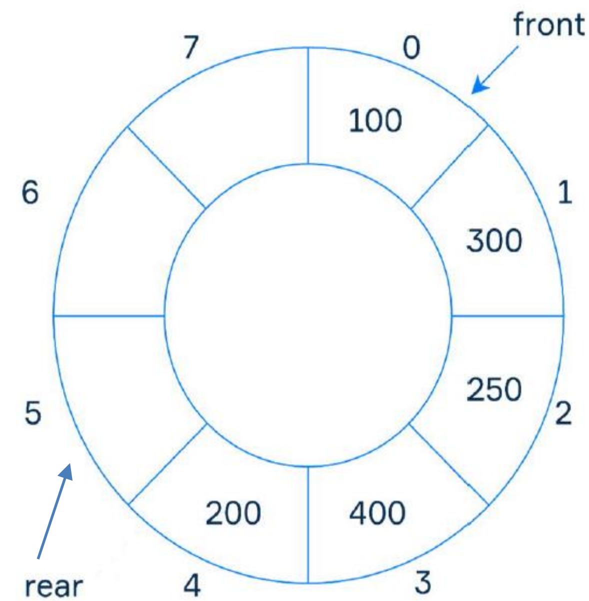
Let f and r wraparound the end of queue (circular queue).



Queues: An Array Implementation

- "wrapped around" configuration





- Each time r or f is incremented, compute this increment as r = (r+1)mod N or f = (f+1)mod N
- When array is full f = r (same as empty condition), then how do we know if it's a queue full or queue empty ? There are many ways to handle this (Explore this!!). Simple approach to handle this, insist that Q can never hold more than N-1 objects (This is your textbook way but not a good way ☹).

Queues: An Array Implementation

Pseudo code

```
Algorithm size()
return (N-f+r) mod N

Algorithm isEmpty()
return (f=r)

Algorithm front()
if isEmpty() then
    return Error
return Q[f]
```

```
Algorithm dequeue()
if isEmpty() then
    return Error
o=Q[f]
Q[f]=null
f=(f+1)mod N
return o
```

```
Algorithm enqueue(o)
if size() = N - 1 then
    return Error
Q[r]=o
r=(r +1)mod N
```

Applications of Queue

- Used in CPU scheduling
- Used in Disk scheduling
- Priority queue is used in heaps [We will look at this in later session]
- …
- …

Arrays: Pluses and minuses

\+ Fast element access.

-- Impossible to resize.

- Many applications require resizing!
- Required size not always immediately available.

List ADT

- A sequence of items where positional order matter
- <a1, a2,…, an-1, an>
- Lists are very general in computing
- e.g. student list, list of events, list of appointments etc

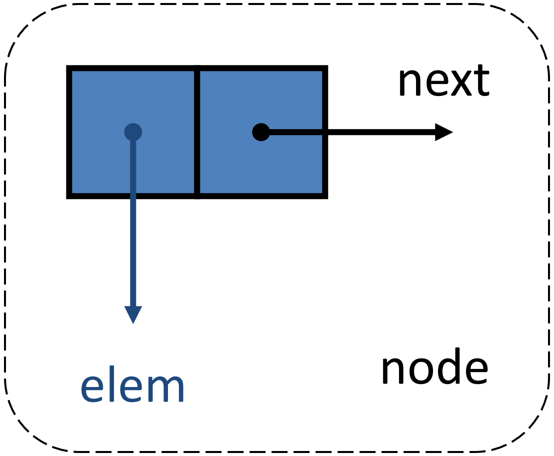
List Operations

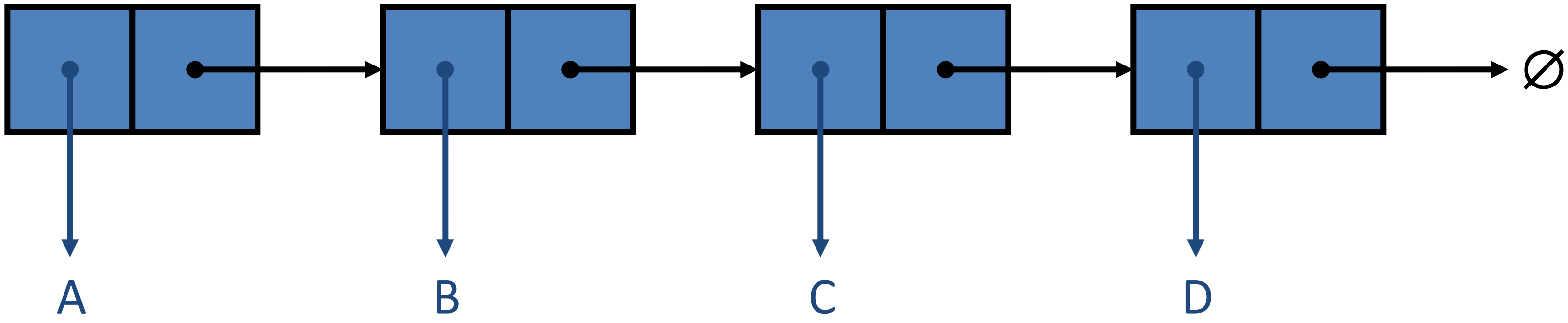
- Supports the following methods for a list S:refer to relative positions in the list
- first ( ):Return the position of the first element of S; an error occurs if S is empty.
- last() :Return the position of the last element of S; an error

  occurs if S is empty.
- isFirst(p) :Return a Boolean value indicating whether the given position is the first one in the list.
- islast (p)
- before(p) :Return the position of the element of S preceding the one at position p; an error occurs if p is the first position.
- after (p) ,size(),isEmpty(),insertAfter(), remove()

Singly Linked Lists

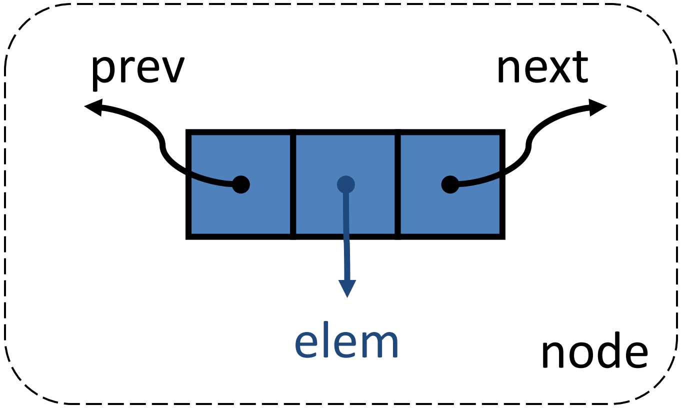
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

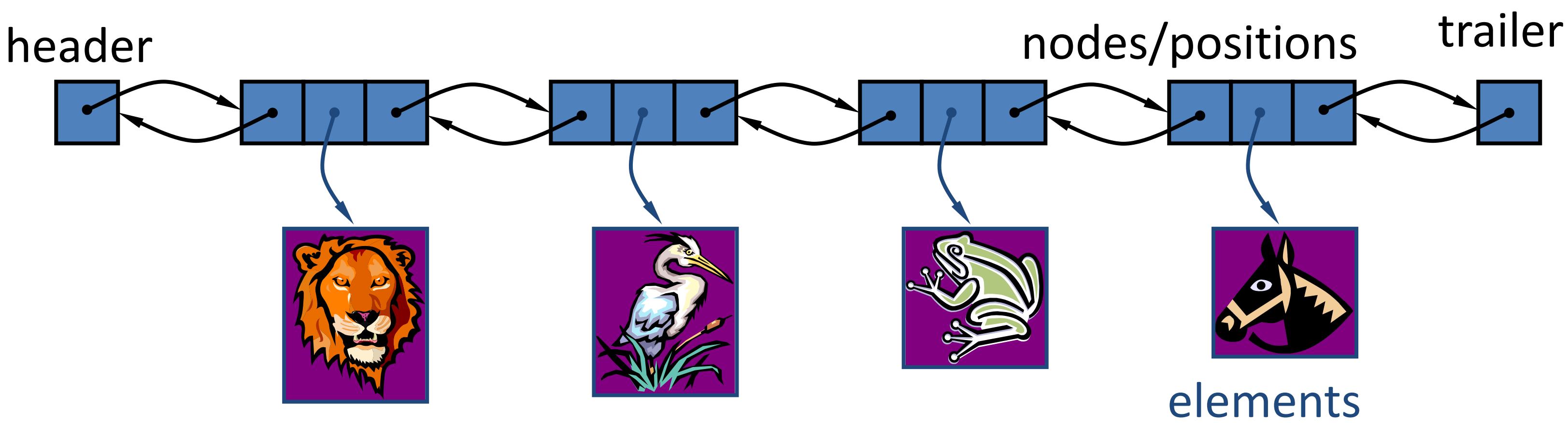




Doubly Linked List

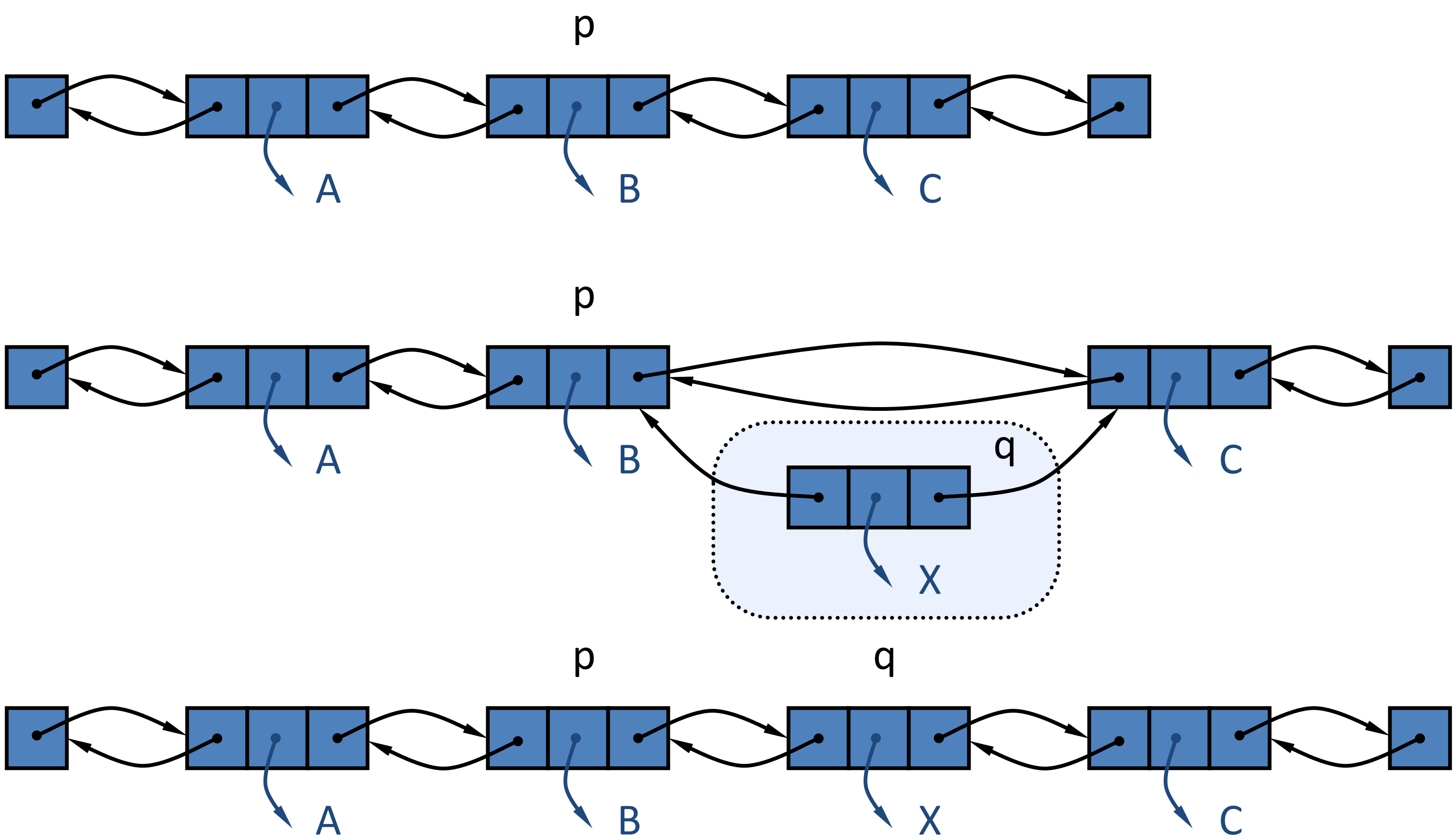
- A doubly linked list is often more convenient!
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

We visualize operation insertAfter(p, X) :



Insertion Algorithm

Algorithm insertAfter(*p*,*e*):

Create a new node *v*

v.setElement(*e*)

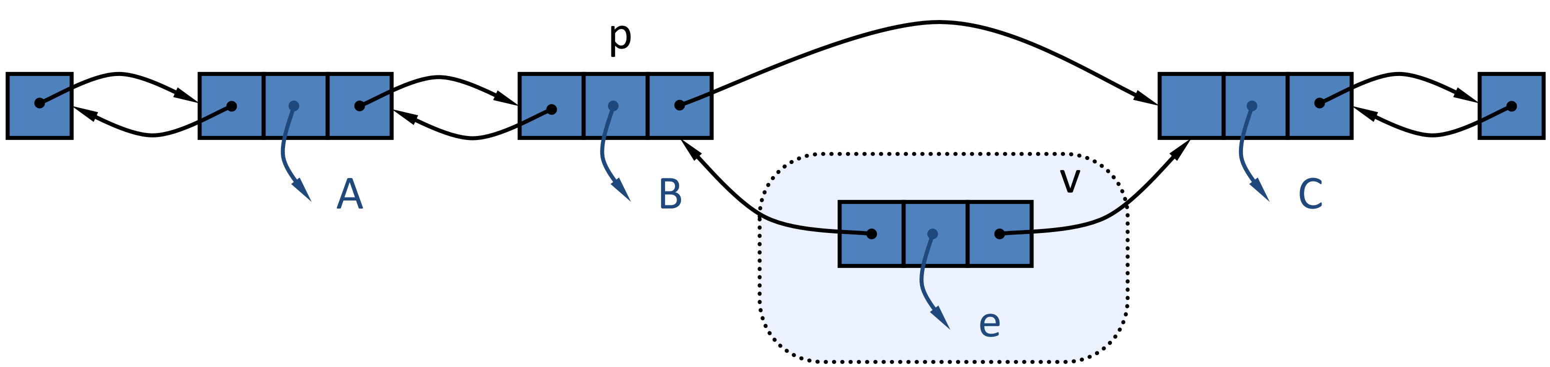
v.setPrev(*p*) {link *v* to its predecessor}

v.setNext(*p*.getNext()) {link *v* to its successor}

(*p*.getNext()).setPrev(*v*) {link *p*'s old successor to *v*}

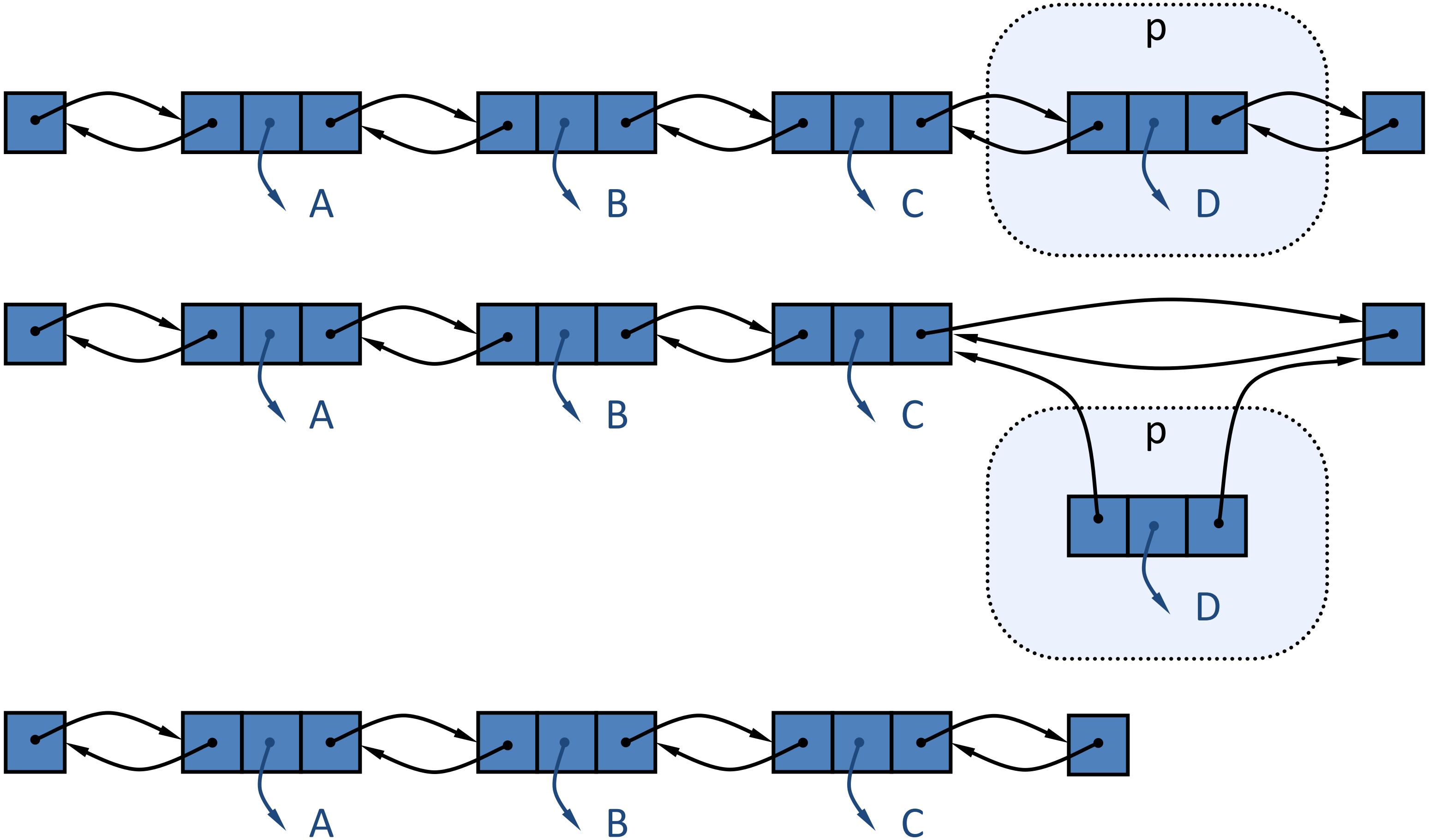
p.setNext(*v*) {link *p* to its new successor, *v*}

return *v* {the position for the element *e*}



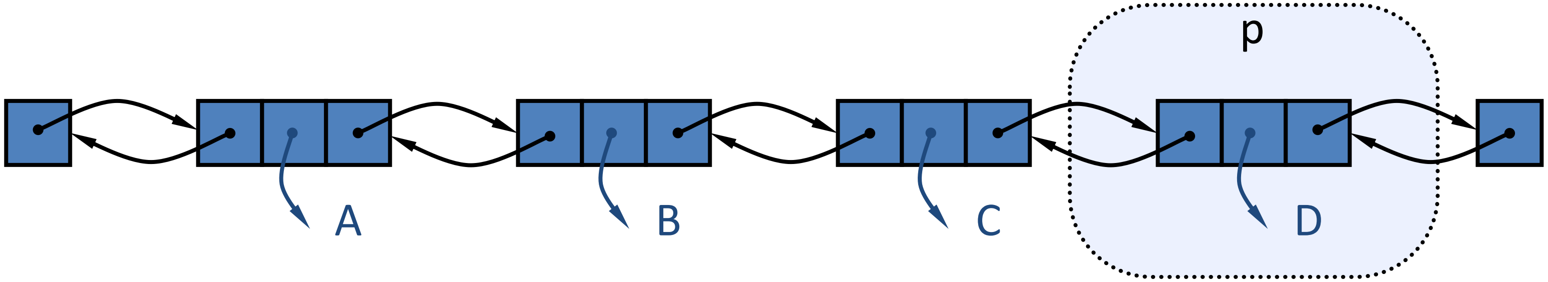
Deletion

- We visualize remove(p)



Deletion Algorithm

Algorithm remove(p):
  $t = p$.element {a temporary variable to hold the return value}
  (p.getPrev()).setNext(p.getNext()) {linking out p}
  (p.getNext()).setPrev(p.getPrev()) {linking out p}
  p.setPrev(**null**) {invalidating the position p}
  p.setNext(**null**)
  return t



Worst-cast running time

In a doubly linked list

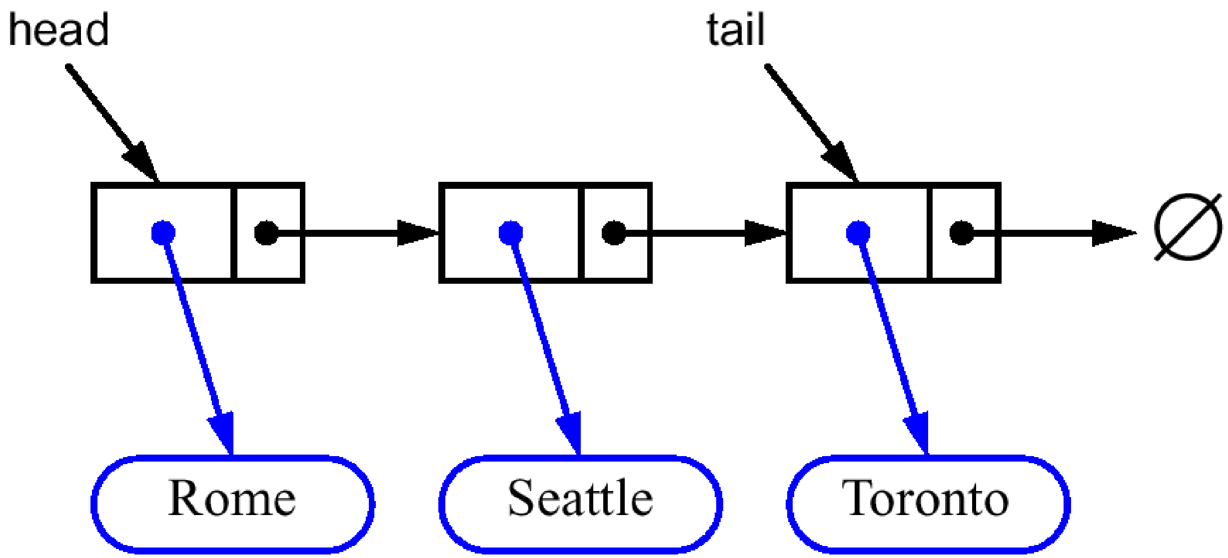
+ insertion at head or tail is in O(1)

+ deletion at either end is on O(1)

-- element access is still in O(n)

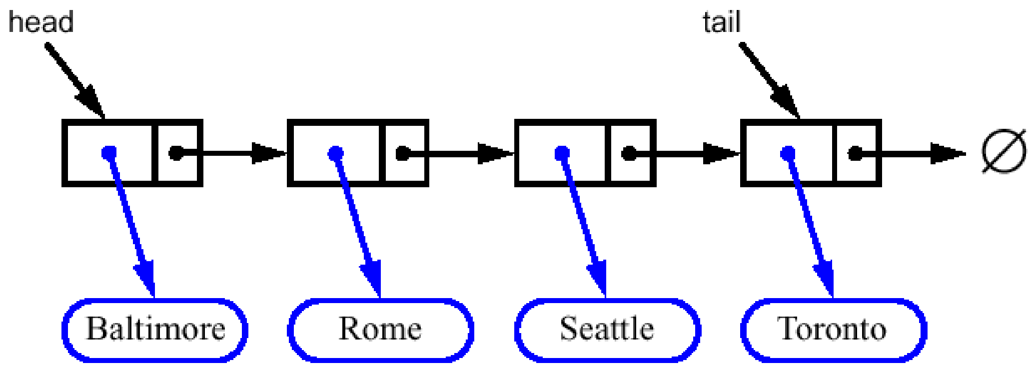
Stacks: Singly Linked List implementation

- Nodes *(data, pointer)* connected in a chain by links

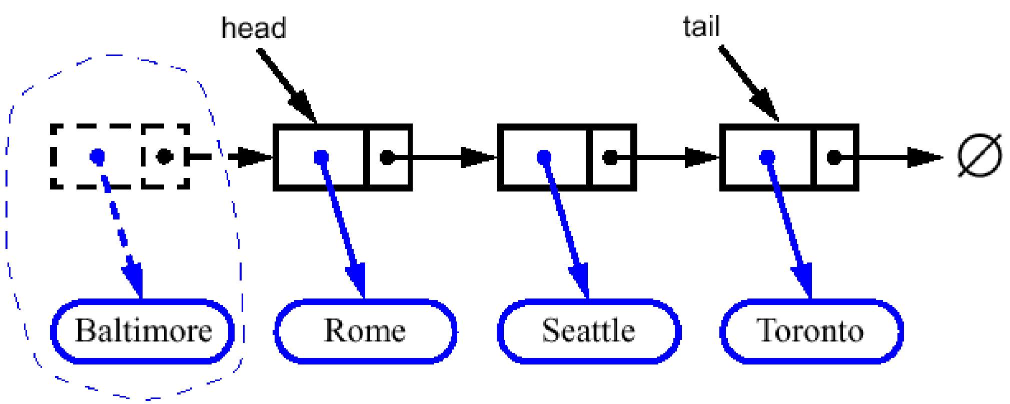


- the head or the tail of the list could serve as the top of the stack

Queues: Linked List Implementation



- Dequeue - advance head reference



Appendix : Forms of Expression

Examples :

#	Infix	Prefix	Postfix
1	(A+B) * C	*+ABC	AB+C*
2	A + (B * C)	+A*BC	ABC*+
3	A ^ B * C – D	- * ^ ABCD	AB ^ C * D -
4	(A+B)/(C-D)	/+AB-CD	AB+CD-/

- The computer usually evaluates an arithmetic expression in infix notation in 2 steps. First, it converts the expression to postfix notation, and then it evaluates the postfix expression. In both these steps, the stack is used !! Let us examine how stack is used ☺

Appendix : Algorithm: Infix to Postfix using Stack

The following algorithm transforms the infix expression Q into its equivalent postfix expression P.

1) Push "(" onto STACK, and add ")" to the end of Q

2) scan Q from left to right and repeat steps 3 to 6 for each element of Q until the stack is empty

3) if an operand is encountered add it to P

4) if a left parenthesis is encountered push it onto the stack

5) if an operator is encountered , then

   (a) Repeatedly pop from STACK and add to P each operator (on top of the STACK) which has same precedence as or higher precedence than the operator encountered

   (b) Add the encountered operator to the stack

   [end of IF structure]

6) if a right parenthesis is encountered, then:

   (a) repeatedly pop from the stack and add to P each operator (on top of the STACK) until a left parenthesis is encountered

   (b) remove the left parenthesis [do not add left parenthesis to P]

   [End of IF structure]

   [End of step 2 loop]

7) Exit

Appendix : Example : Converting Infix to Postfix using Stack

Consider the infix expression Q to be **A + B * C → A+B*C)**

#	Symbol Encountered	Postfix String P (Output)	STACK Contents
1			(
2	A	A	(
3	+	A	(+
4	B	AB	(+
5	*	AB	(+*
6	C	ABC	(+*
7	)	ABC*	(+
8	-	ABC*+	(
9	-	ABC*+	Empty

Hence, the postfix expression P is **ABC*+**

Appendix : Example : Converting Infix to Postfix using Stack

Consider the infix expression Q to be (A+B) * C ➔ (A+B) * C)

#	Symbol Encountered	Postfix String P (Output)	STACK Contents
1			(
2	(		((
3	A	A	((
4	+	A	((+
5	B	AB	((+
6	)	AB+	(
7	*	AB+	(*
8	C	AB+C	(*
9	)	AB+C*	Empty
10	-	AB+C*	Empty

Hence, the postfix expression P is **AB+C***

Appendix Exercise

1) Convert the following infix expressions into postfix expressions using stack (trace it as we did in class) :

- A+(B+C)*D
- B+C-D/E*F
- (A+B)+(C/D)*E
- A+B*(C+D-E)*F

After you solve it, verify your answer by putting the infix expression in this tool : https://www.mathblog.dk/tools/infix-postfix-converter/

Thank You for your time & attention !

Contact : parthasarathypd@wilp.bits-pilani.ac.in