2. Lists, Stacks and Queues (Continued)

DATA STRUCTRES AND ALGORITHMS
[17ECSC204]

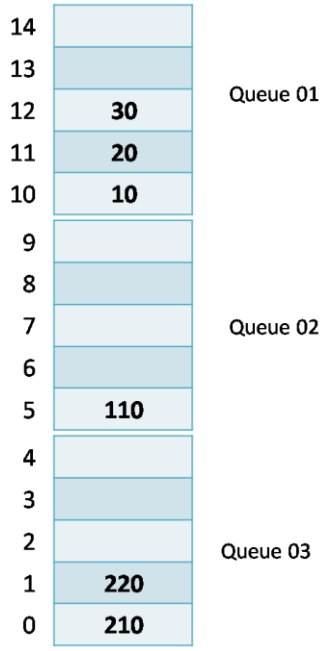
PRAKASH HEGADE

SCHOOL OF CSE | KLE Tech

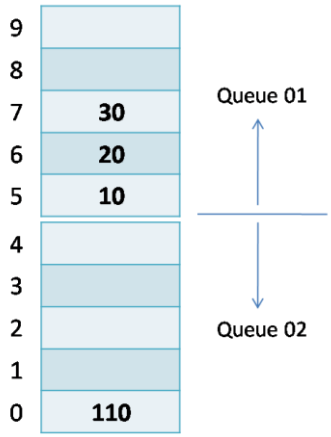
Multiple Queues

Multiple Queues can be
1. Multiple FIFO Queue with statically partitioned space
2. Multiple FIFO Queue with dynamically shared Space

Statically portioned space is where size of the each queue is pre-known. The memory is partitioned and allocated for each queue. Each queue implementation can be of type as demanded by application needs (Linear, circular etc).



Dynamically shared space is where the memory allocated for each queue keeps varying. Based on dynamic requests, the size is readjusted.

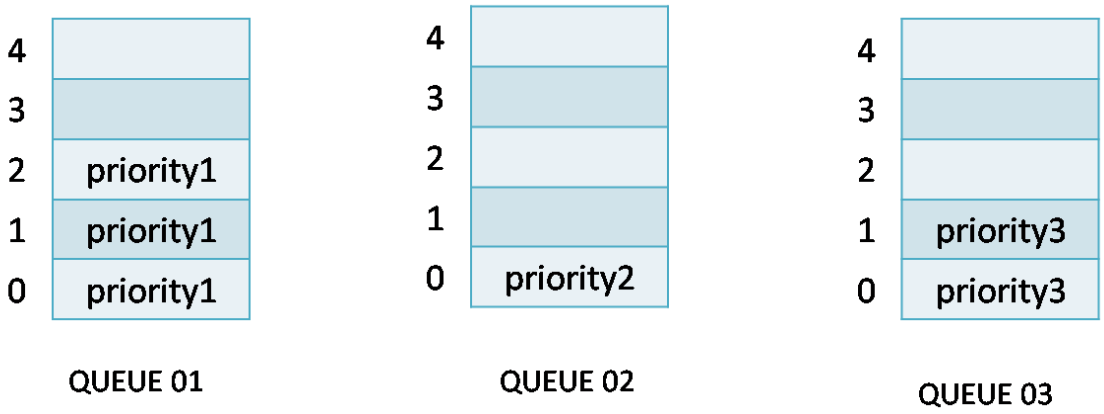


The type of queue will be selected based on the nature of the problem. The challenges and Issues with Multiple Queues are:

- Space management
- Improving the time factor with multiple queues

We can also visualize multiple queues as separate queue for each task. Instead of lining up all tasks in same queue we can classify and put jobs in separate queues and a separate processor for each queue. As an example you can remember bank counters. If there was only one counter in bank to manage all kind of jobs, then work would have been slow and tedious. As they have different counter for different type of tasks, the work is distributed and carried out faster.

As an example you can see below, multiple queues where each queue is used to process the same priority jobs. Queue 02 is executed only when Queue 01 become empty. And internally each queue processing discipline can be of type as application demands (Linear, Circular, and Priority etc).



Application of Queues

Because of its first in first out nature, Queue finds application in many areas.

The basic problem can be viewed as,
In every problem there is a producer and consumer. The production produced by producer is consumed by consumer. If Producer is producing at a very faster rate and consumer has slower consumption rate then we would need an intermediate storage medium to store all the produced items. As the first produced item needs to be consumed first, we would need a **queue.**

Producer < - - > QUEUE < - - > Consumer

1) Imagine you have a web-site which serves files to thousands of users. You cannot service all requests; you can only handle say 100 at once. A fair policy would be first-come-first serve: serve 100 at a time in order of arrival. A Queue would definitely be the most appropriate data structure.

2) When a resource is shared among multiple consumers we need a queue
Examples include:

- CPU scheduling that is the way your processor schedules different tasks
- Disk Scheduling (you will study this in your Operating System course).

3) Say you have a number of documents to be printed at once. Your printer software puts all of these documents in a queue and sends them for print job. The printer takes and prints each document in the order they are put in the queue, ie, First In, First Out.

In the situation where there are multiple users or a networked computer system, you probably share a printer with other users. When you request to print a file, your request is added to the print queue. When your request reaches the front of the print queue, your file is printed.

4) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes we would need a queue to store.
Examples include IO Buffers, pipes, file IO, etc. (you will study this in your Operating System and Networking courses)

More Programs on Linked List

1. How do you find out if one of the pointers in the linked list is **corrupted or not**?

- Frequent consistency checks
- Set pointer to NULL immediately after freeing the memory pointed by pointer
- Use good tools, debuggers
- Avoid global variables
- Testing for all possible scenarios in code
- Review code well
- Use a extra node field to keep number of nodes after his node
- Keep the count of total number of nodes

2. **Header Nodes:**
Generally in linked list implementations we use an extra starting node called as header node. Instead of start pointer we can use a node itself and this can be used to keep list statistics like total number of nodes and other information.

3. **Delete a node** from Linked list
Alternate Solution:
Copy the data from next node to this node and delete next node

4. **Sort** a linked list

- List can be sorted as it is built (Insert at appropriate position)
- Sorting techniques like bubble sort can be applied on the linked list

5. **Reverse** a linked list

- Keep the list as it is, swap the data (swapping the data from first and last node and so on)
- Create a new list with reverse data

6. How do you detect a **loop** in a linked list?

```
NODE * find_loop(NODE * start)
{
        NODE *current = start;
        while(current->next != NULL)
        {
                NODE *temp = start;
                while(temp->next != NULL && temp != current)
```

```
                {
                        if(current->next == temp)
                        {
                                printf("\n Loop Detected..\n");
                                return current;
                        }
                        temp = temp->next;
                }
                current = current->next;
        }
        return NULL;
}
```

7. Create a **Copy** of linked list
Traverse the list till end and with respect to every node call the function
insert_at_end( )

8. How do you find **Middle node** in the linked list?
"The fast pointer – slow pointer technique"
In every iteration, increment fast pointer by two nodes and slow pointer by one node. When fast pointer reached last node, slow pointer will be pointing to middle node in the list.

```
void Compute(NODE * start)
{
  NODE *p = start, *q = start;
  if(q!=NULL)
  {
    while((q->next)!=NULL && (q->next->next)!=NULL)
    {
      p = ( p != NULL ? p->next : NULL);
      q = ( q != NULL ? q->next : NULL);
      q = ( q != NULL ? q->next : NULL);
    }
    printf("The middle node of the list is [%d]",p->info);
  }
}
```

9. **Compare** 2 linked lists

```
int compare_linked_lists(NODE *q, NODE *r)
{
  static int flag;
  if((q==NULL ) && (r==NULL))
    flag=1;
  else
  {
    if(q==NULL || r==NULL)
      flag=0;
    if(q->info!=r->info)
      flag=0;
```

```
      else
         compare_linked_lists(q->next,r->next);
   }
   return(flag);
}
```

10. **Binary search** on linked lists

The pre-condition is that the linked list has to be ordered. (Elements present in sorted fashion). Until the key is found, the mid of the linked list has to be obtained. Sequential search performs better on linked lists than binary search.

11. **Concatenate** two linked lists

```
NODE * concatenate_two_lists (NODE * first, NODE * second)
{
   NODE * temp;
   if(first == NULL && second == NULL)
     return first;
   if(first == NULL)
      return second;
   else if (second == NULL)
      return first;

   // Obtain the address of last node of first list
   temp = first;
   while(temp->next != NULL)
      temp = temp->next;

   // Attach the first node of second list to end of first list
   temp->next = second;

   // Return the starting address of first list
   return first;
}
```

12. **Reverse** a linked list

```
NODE * reverse (NODE * first)
{
   NODE * current, *temp;
   current = NULL;

   while(first!= NULL) {
      temp = first;
      first = first->next;
      temp->next = current;
      current = temp;
   }
   return current;
}
```

13. Finding the **size of structure**: [simulates sizeof()]

```
struct node {
   int data;
   struct node * next;
};
int main() {
   struct node *n;
   char * x1 = (char *)(n + 1);
   char * x2 = (char *) n;
   int x = x1 - x2;
   printf("\nRESULT : [%d]\n", x);
   return 0;
}
```

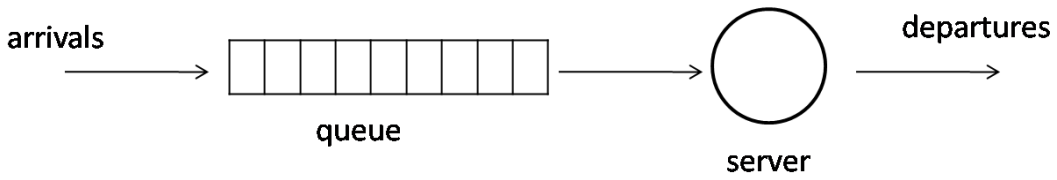
14. How would you?

- Remove the duplicates from the linked list?
- Read a Singly linked list backwards?
- Search a data in linked list?
- Count the number of nodes in linked list?

M/M/1 Queue

Queueing theory provides a mathematical basis for understanding and predicting the behaviour of communication networks.

Basic Model:



We can understand above model with below mentioned parameters:

- Inter-arrival time distribution
- Service time distribution
- Number of servers
- Queueing disciplines
- Number of buffers

Common Notion:

A/B/m
Where: 'm' is the number of servers
A and B can be from
M: Markov (exponential distribution / Poisson)
D: Deterministic
G: General (arbitrary distribution)

M/M/1 stands for:

- Inter-arrival times are Poisson distributed (exponentially)
- Service times are exponentially distributed
- There is only one server
- The buffer is assumed to be infinite
- Queueing discipline is FCFS

Infix, Postfix and Prefix Expressions

Note: This is summary notes only. Please refer class notes for more.

Expressions can be classified as infix, prefix and postfix.
Consider the sum of A and B,
A + B → Infix
+ A B → prefix
A B + → postfix

Position of operator with respect to two operands define infix, prefix and postfix respectively.

We consider 5 binary operators

+, -, *, /, $ (exponentiation)

The order of precedence is

$
*, /
+, -

Associativity

*, /, + , - → left to right
$ → right to left

Before converting the expression from infix to prefix and postfix, we need to first parenthesize them. Example to parenthesize expressions:

A + B + C	→	(A + B) + C
A $ B $ C	→	A $ (B $ C)
A * B * C + D	→	(((A * B) * C) + D)
(A + B) * (C - D)	→	((A + B) * (C - D))

Examples for Infix to Postfix Conversion

Infix	Parenthesize	Postfix
(A + B) * (C - D)	((A + B) * (C - D))	AB + CD - *
A $B * C – D + E / F / (G + H)	(((A $ B) * C) – D) + ((E / F) / (G + H))	AB $ C * D – EF / GH + / +
((A + B) * C – (D - E)) $ (F + G)	(((A + B) * C)– (D - E)) $ (F + G))	AB + C *DE - -FG+ $
A – B / (C * D $ E)	(A – (B / (C * (D $ E))))	ABCDE$*/-

Examples for Infix to Prefix Conversion:

Infix	Parenthesize	Prefix
(A + B) * (C - D)	((A + B) * (C - D))	*+ AB - CD
A $B * C – D + E / F / (G + H)	(((A $ B) * C) – D) + ((E / F) / (G + H))	+ - * $ ABCD // EF + GH
((A + B) * C – (D - E)) $ (F + G)	(((A + B) * C)– (D - E)) $ (F + G))	$ - * + ABC – DE + FG
A – B / (C * D $ E)	(A – (B / (C * (D $ E))))	- A / B * C $ DE

Procedure to Evaluate a Postfix Expression

1. Scan the input and read the expression symbols one by one.
2. If the symbol is operand, push it into the stack
3. If the symbol is operator, pop the two values from stack where the first popped symbol is operand number 2, apply the operator on two popped operands and push the result back to stack

Example:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

symbol	opnd1	opnd2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3				1,3
8				1,3,8
2				1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2				7,2
$	7	2	49	49
3				49,3
+	49	3	52	52

Algorithm:

```
opndstk = empty
/* scan the inut string reading one element at a time into symb */
while(not end of input) {
   symb = next input character
   if (symb is an operand)
      push(opndstk, symb)
   else {
      opnd2 = pop(opndstk)
      opnd1 = pop(opndstk)
      value = result of applying symb to opnd1 and opnd2
      push(opndstk, value)
   }
}
return (pop(opndstk))
```

Converting an Infix to Postfix Expression

Rule:
prcd(op1, op2) : TRUE, if op1 has higher precedence over op2
[when op1 appears to left of op2 in an infix expression without parenthesis]
FALSE, otherwise

Examples:
prcd(* , +) – TRUE
prcd(+ , +) – TRUE
prcd(+ , *) – FALSE
prcd($, $) – FALSE

When FALSE we push to stack. If TRUE, we add op1 to string and push op2 to stack.

Let us apply on:
A+ B * C $ D $ E

symb	postfix string	opstk
A	A	
+	A	+
B	AB	+
*	AB	+*
C	ABC	+*
$	ABC	+*$
D	ABCD	+*$
$	ABCD	+*$
E	ABCDE	+*$$
	ABCDE$$*+	

Property: at any given time, the elements below stack top are with lower precedence

Example 02: A + B – C

symb	postfix string	opstk
A	A	
+	A	+
B	AB	+
-	AB	+
	AB +	
	AB+	-
C	AB+C	-
	AB+C-	

Algorithm:
opstk = the empty stack;
while(not end of input) {
  symb = next input character;

```
  if(symb is an operand)
    add symb to the postfix string
  else
  {
    while(!empty(opstk) && prcd(stacktop(opstk), symb)) {
      topsymb = pop(opstk);
      add the topsymb to the postfix string;
    }
    push(opstk, symb);
  }
}

while(!empty(opstk)) {
  topsymb = pop(opstk);
  add topsymb to the postfix string;
}
```

Now, considering the expressions with parenthesis, we have:
When an opening parenthesis is read it must be pushed into the stack.
So, **prcd(op, '(')** → FALSE, for op any operator other than ')'.

So with the case when any operator is read and '(' is top of thr stack.
prcd('(', op) → FALSE

When a closing parenthesis is read, all the operators up to the first opening parenthesis must be popped from the stack into the postfix string.
prcd(op, ')') → **TRUE** and the opening parenthesis along with closing is discarded.
prcd('(' , ')') → **FALSE**

Summarizing:
prcd(op, '(') → FALSE, for op any operator other than ')'.
prcd('(', op) → FALSE for any op
prcd(op, ')') → TRUE for any operator other than '('
prcd(')', op) → undefined

Example: (A + B) * C

symb	postfix string	opstk
(		(
A	A	(
+	A	(+
B	AB	(+
)	AB +	
*	AB +	*
C	AB+C	*
	AB + C *	

Example: ((A – (B + C)) * D) $ (E + F)

symb	postfix string	opstk
(		(
(		((
A	A	((
-	A	((-
(	A	((- (
B	AB	((- (
+	AB	((- (+
C	ABC	((- (+
)	ABC+	((-
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	
$	ABC+-D*	$
(	ABC+-D*	$ (
E	ABC+-D*E	$ (
+	ABC+-D*E	$ (+
F	ABC+-D*EF	$ (+
)	ABC+-D*EF+	
	ABC+-D*EF+$	

Algorithm:

```
opstk = the empty stack;
while(not end of input) {
   symb = next input character;
   if(symb is an operand)
      add symb to the postfix string
   else
   {
      while(!empty(opstk) && prcd(stacktop(opstk), symb)) {
      topsymb = pop(opstk);
      add the topsymb to the postfix string;
      }
      if(empty(opstk) || symb != ')')
           push(opstk, symb);
      else
           topsymb = pop(opstk); // pop the '(' and discard it
   }
}

while(!empty(opstk)) {
   topsymb = pop(opstk);
   add topsymb to the postfix string;
}
```

Puzzles / Problems Discussed in Class

1. The Celebrity Problem
Only one person is known to everyone. And that person may or may not be present in the party.

Hypothetical function: *HaveAcquanitance(A,B)* is true if A knows B and false otherwise.

We can ask only one question – Does A know B?

- If A knows B then A can't be celebrity. Discard A and B may be celebrity
- If A does not know B then B can't be celebrity. Discard B and A may be celebrity
- Ensure remained person is celebrity

2. Queue Interleaving
For a queue of integers of even length, rearrange by interleaving first half of queue with second half.

Example:
Input: 1 2 3 4 5 6
Output: 1 4 2 5 3 6

Steps:

- Push the first half elements of queue to the stack
- Enqueue back the elements from the stack
- Dequeue the first half elements from queue and enqueue them back to queue
- Again push the first half elements into the stack
- By above steps, we will have half into stack and half into queue. Now interleave the elements by picking one from stack and one from queue and enqueuer-ing them back into queue

3. Reverse A Queue
Reverse a queue using a stack.
Dequeue and Push into stack and then pop and enqueue back into queue

4. Reverse the first K elements of queue

- Send the first K elements of queue to stack
- Pop from stack and enqueue to queue
- Dequeue and enqueue the remaining elements into queue

5. Track current maximum element from stack
Example:
Input: 4 19 7 14 20
Output: 4 19 19 19 20

- Keep two stacks. One operative stack called 'OperStack' for elements and one 'maxStack' to keep track of maximum element.

- As the first element comes in, push it into both stacks.
- For all the remaining elements which come in, compare it with top most element of 'maxStack'. If it is greater than top most elements of 'maxStack' then push it onto both the stacks, otherwise push it only into 'operStack' and push the same top most element of 'maxStack' once again into it.

6. Generate binary numbers from 1 to n using queue

Input: n = 3
Output: 1, 10, 11

Input: n= 5
Output: 1, 10, 11, 100, 101

- Initialize queue with 1
- Repeat this step until required 'n' is reached.
    - Dequeue front and print
    - Suffix front element with '0' once and '1' another and enqueue both into queue in the same order

~*~*~*~*~*~*~