Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

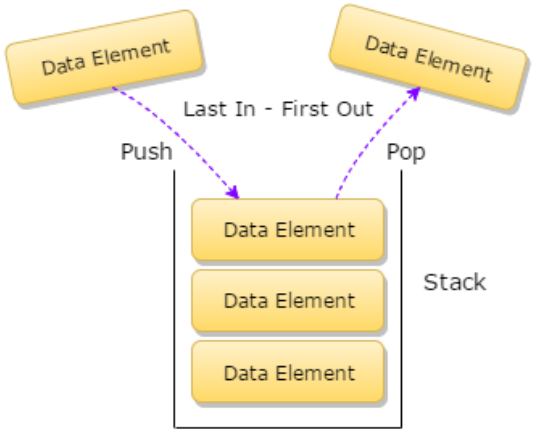
Stacks

Dinesh Singh
Department of Computer Science & Engineering

Data Structures and its Applications

Stacks - Definition

- A Stack is a data Structure in which all the insertions and deletions of entries are at one end. This end is called the TOP of the stack.
- When an item is added to a stack it is called push into the stack
- When an item is removed it is called pop from the stack.
- The Last item pushed onto a stack is always the first that will be popped from the stack.
- This property is called the *last in, first out* or LIFO for short



Stacks – Representation in C

A stack in C is declared as a structure containing two objects :

- An array to hold the elements of the stack
- An Integer to indicate the position of the current stack top within the array
- Stack of integers can be done by the following declaration

```
#define STACKSIZE 100
struct stack
{
  int top;
  int items[STACKSIZE]
};
```

Once this is done, actual stack can be declared by

```
struct stack s;
```

Stacks – Representation in C

Items need not be restricted to integers, items can be of any type.

A stack can contain items of different types by using C unions.

```
#define STACKSIZE 100
#define INT 1
#define FLOAT 2
#define STRING 3
struct stackelement {
    int etype;
    union{
        int ival;
        float fval;
        char *pavl; //pointer to string
    } element;
};
```

Stacks – Representation in C

```
struct stack
{
  int top;
  struct stackelement items[STACKSIZE];
};
```

- The above declaration defines a stack whose items can either be integers, floating point numbers or string depending on the value of etype (previous slide).

Data Structures and its Applications

Stacks – Implementation of operations of stack

Operations on stack

- Inserting an element on to the stack : push
- Deleting an element from the stack : pop
- Checking the top element : peep
- Checking if the stack is empty : empty
- Checking if the stack is full : overflow

Representation of stack will be as follows

```
#define STACKSIZE 100
struct stack
{
  int top;
  int items[STACKSIZE]
};
```

Stacks – Implementation of operations of stack

```
void push(struct stack *ps, int x)
/*ps is pointer to the structure representing stack, x is integer to be inserted
top is integer that indicates the position of the current stack top within the
      array, items is an integer array that represents stack, STACK_SIZE is the
      maximum size of the stack */
{
   if (ps->top == STACKSIZE -1) //check if the stack is full
         printf("STACK FULL Cannot insert..");
  else
   {
      ++(ps->top); //increment top
      ps->items[ps->top]=x; //insert the element at a location top
    }
}
```

Data Structures and its Applications

Stacks – Implementation of operations of stack

```
int pop(struct stack *ps )
/*ps is pointer to the structure representing stack, top is integer that indicates
      the position of the current stack top within the array , items is an integer
      array that represents stack, STACK_SIZE is the maximum size of the stack */
  {
   if (ps->top == -1) // check if the stack is the empty
          printf("STACK EMPTY Cannot DELETE..");
  else
   {
      x=ps->items[ps->top]; //delete the element
      --(ps->top); //decrement top
     return x;
    }
}
```

Data Structures and its Applications

Stacks – Implementation of operations of stack

```
int display(struct stack *ps )
/*ps is pointer to the structure representing stack, top is integer that indicates
      the position of the current stack top within the array , items is an integer
      array that represents stack, STACK_SIZE is the maximum size of the stack */
  {
   if (ps->top == -1) // check if the stack is the empty
          printf("STACK EMPTY ");
  else
   {
      for (i=ps->top;i>=0;i--) // displays the elements from top
         printf("%d",ps->items[i]);
 }
}
```

Stacks – Implementation of operations of stack

```
int peep(struct stack *ps )
{
   if (ps->top == -1)
          printf("STACK EMPTY ..");
  else
   {
      x=ps->items[ps->top]; //get the element
      return x;
    }
}
```

Stacks – Implementation of operations of stack

```
int empty(struct stack *ps )
{
    if (ps->top == -1)
            return 1;
   return 0;
}
int overflow(struct stack *ps)
{
    if (ps->top==STACKSIZE-1)
      return 1;
  return 0;
}
```

Stacks – Implementation of operations of stack

implementation of stack operations where the items array and top are separate variables (Not part of structure)

```
void push(int *s, int *top, int x)
{
  if(*top==STACKSIZE-1)//check if the stack is full
  {
    printf("stack overflow..cannot insert");
   return 0;
 }
 else
  {
    ++*top; //increment the top
    s[*top]=x; //insert the element
  }
 return 1;
}
```

Stacks – Implementation of operations of stack

- Implementation of stack operations where the items and the top are separate variables (Not part of structure)

```
int pop(int *s, int *top)
{
  if(*top==-1)//check if the stack is empty
  {
    printf("Stack empty .. Cannot delete");
   return -1;
 }
 else
  {
    x=s[*top]; //insert the element
    --*top; //decrement top
   return x; // return the deleted element
  }
}
```

Stacks – Implementation of operations of stack

- Implementation of stack operations where the items and the top are separate variables (Not part of structure)

```
display(int *s, int *top)
{
   if(*top==-1)
    printf("Empty stack");
 else
  {
   for(i=*top;i>=0;i--) // display the elements from the top
     printf("%d",s[i]);
  }
}
```

Stacks – Application of Stack

- Write an algorithm to determine if an input character string is of the form x C y where x is a string consisting of the letters 'A' and 'B' and where y is the reverse of x. At each point you may read only the character of the string

```
 int check(t)
//the function returns 1 if string t is of the form x C y, else returns 0
//uses stack s and its operations push and pop
 {
    i=0;
    while(t[i]!='C') //push all the characters of the string into the stack
 until C  is encountered
    {
      push(&s, t[i]);
      i=i+1;
    }
```

```
      //pop the contents of the stack and compare with t[i] until the end of  the string
while(t[i]!='\0')
{
   x=pop(&s);
   if(t[i]!=x) //if the character popped out is not equal to the character read from the string
      return 0; // not of the form xCy
   i=i+1;
 }
 return 1; // string of the form
 }
```

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Stacks – Linked List Implementation

Dinesh Singh
Department of Computer Science & Engineering

Data Structures and its Applications

Stacks – linked list implementation

- A stack can be easily implemented through the linked list. In the Implementation by a linked list, the stack contains a top pointer. which is “head” of the stack. The pushing and popping of items happens at the head of the list.

Structure of the stack

```
struct node
{
   int data;
   struct node *next;
 }
struct node *top
top=NULL;
```

Data Structures and its Applications

Stacks – linked list implementation

Note :

- Items of the stack represented as the linked list.
- Each item is a node
- Top is a pointer that points to the first node (top of the stack)
- Top is initially NULL (Empty stack)
- Insertion and deletion happens at the front of the list
- stack size is not limited.

Operations on the stack

- Push : Inserting an element at the front of the list
- Pop : delete an element from the front of the list
- Display : displaying the list

Stacks – linked list implementation

```
//implements the push operation
void push(int x, struct node **top)
   {
        struct node *temp;
        temp=(struct node*)malloc(sizeof(struct node));
        temp->data=x;
        temp->next=*top; // insert in front of the list
        *top=temp; // make top points to the new top node
    }
```

Stacks – linked list implementation

```
//implements the pop operation
//returns top element, -1 if stack is empty
int pop(struct node **top)
    {
        int x;
       struct node *q;

       if(*top==NULL)
        {
          printf("Empty Stack\n");
           return -1;
         }
```

Stacks – linked list implementation


```
//implements the pop operation

else
        {
          q=*top;
           x=q->data; // get the top element
          *top=q->next; // make top point to the next top
          free(q); // free the memory of the node
           return(x);
         }
    }
```

Stacks – linked list implementation

```
//implements the display  operation
void display(struct node *top)
    {
       if(top==NULL)
         printf("Empty Stack\n");
       else
        {
          while(top!=NULL)
          {
            printf("%d->“,top->data);
            top=top->next;
          }
        }
    }
```

Stacks – linked list implementation

Another representation of structure of stack

```
struct node
{
  int data;
  struct node *next;
};
struct stack
{
   struct node *top;
}

struct stack s;
s.top=NULL;
```

Stacks – linked list implementation

```
//implements the push operation
void push(int x, struct stack * s)
//s is pointer to structure stack
   {
        struct node *temp;
        temp=(struct node*)malloc(sizeof(struct node));
        temp->data=x;
        temp->next=s->top; // insert in front of the list
        s->top=temp; // make top points to the new top node
    }
```

Stacks – linked list implementation

```
//implements the pop operation
//returns top element, -1 if stack is empty
int pop(struct stack *s)
    {
        int x;
       struct node *q;

       if(s->top==NULL)
        {
          printf("Empty Stack\n");
           return -1;
         }
```

Stacks – linked list implementation

```
//implements the pop operation

else
        {
          q=s->top;
           x=q->data; // get the top element
          s->top=q->next; // make top point to the next top
          free(q); // free the memory of the node
           return(x);
         }
    }
```

Stacks – linked list implementation

```
//implements the display  operation
void display(struct stack *s)
   {
      struct node *q;
      if(s->top==NULL)
        printf("Empty Stack\n");
      else
       {
        q=s->top;
        while(q!=NULL)
        {
          printf("%d->“,q->data);
          q=q->next;
         }
       }
   }
```

Stacks – Application

Write an Algorithm to print a string in the reverse order

```
//prints the text in a reverse order
reverse(t)
{
 i=0;
 //push all the characters on to the stack
 while(t[i]!='\0')
 {
    push(s,t[i]);
    i=i+1;
 }
```

Stacks – Application

pop all the characters from the stack until the stack is empty

```
while(!empty(s))
 {
    x= pop(s);
    print(x)
   }
}
```

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Application of stacks– Functions, nested functions and Recursion

Dinesh Singh
Department of Computer Science & Engineering

Application of stacks – Functions, nested functions

Activation record :

- When functions are called, The system (or the program) must remember the place where the call was made, so that it can return there after the function is complete.
- It must also remember all the local variables, processor registers, and the like, so that information will not be lost while the function is working.
- This information is considered as large data structure . This structure is sometimes called the invocation record or the activation record for the function call.

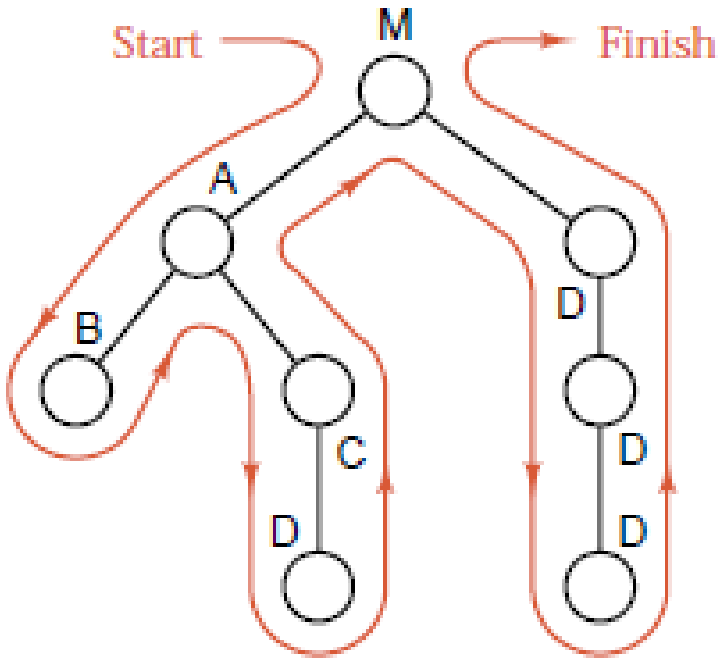
Application of stacks – Functions, nested functions and Recursion

- Suppose that there are three functions called A,B and C. M is the main function.
- Suppose that A invokes B and B invokes C. Then B will not have finished its work until C has finished and returned. Similarly, A is the first to start work, but it is the last to be finished, not until sometime after B has finished and returned.
- Thus the sequence by which function activity proceeds is summed up as the property last in, first out.
- The machine's task of assigning temporary storage area (activation records) used by functions would be in same order (LIFO).
- Since LIFO property is used, the machine allocates these records in the stack
- Hence a stack plays a key role in invoking functions in a computer system.

Application of stacks – Functions, nested functions

Example :

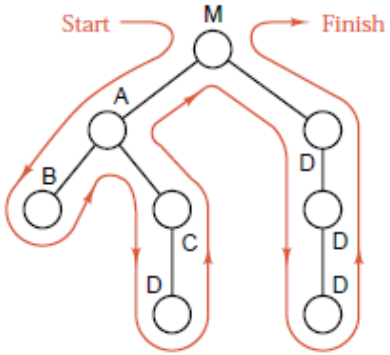
Consider the following showing the order in which the functions are invoked

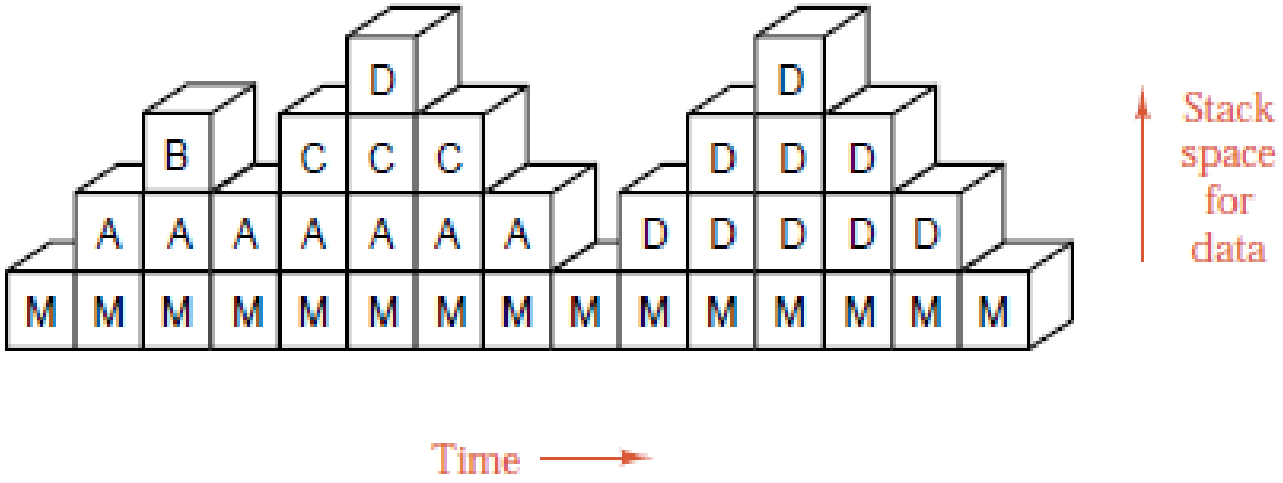


Data Structures and its Applications

Application of stacks – Functions, nested functions

The Sequence of stack frames of the activation records of the function calls is given below. Each vertical column shows the contents of the stack at a given time, and changes to the stack are portrayed by reading through the frames from left to right.





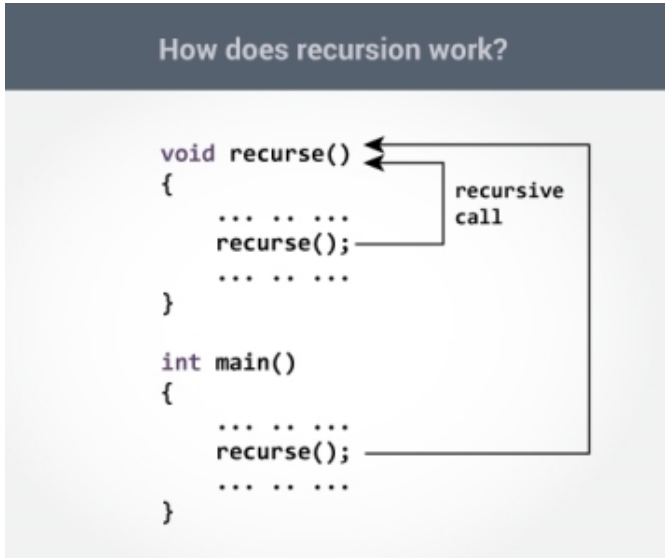
Data Structures and its Applications

Application of stacks – Recursion

Recursion :

- Recursion is a computer programming technique involving the use of a procedure, subroutine or a function.

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.



Application of stacks – Recursion

How a particular problem is solved using recursion?

- The idea is to represent a problem in terms of one or more smaller problems.

- A way is found to solve these smaller problems and then build up a solution to the entire problem.

- The sub problems are in turn broken down into smaller sub problems until the solution to the smallest sub problem is known.

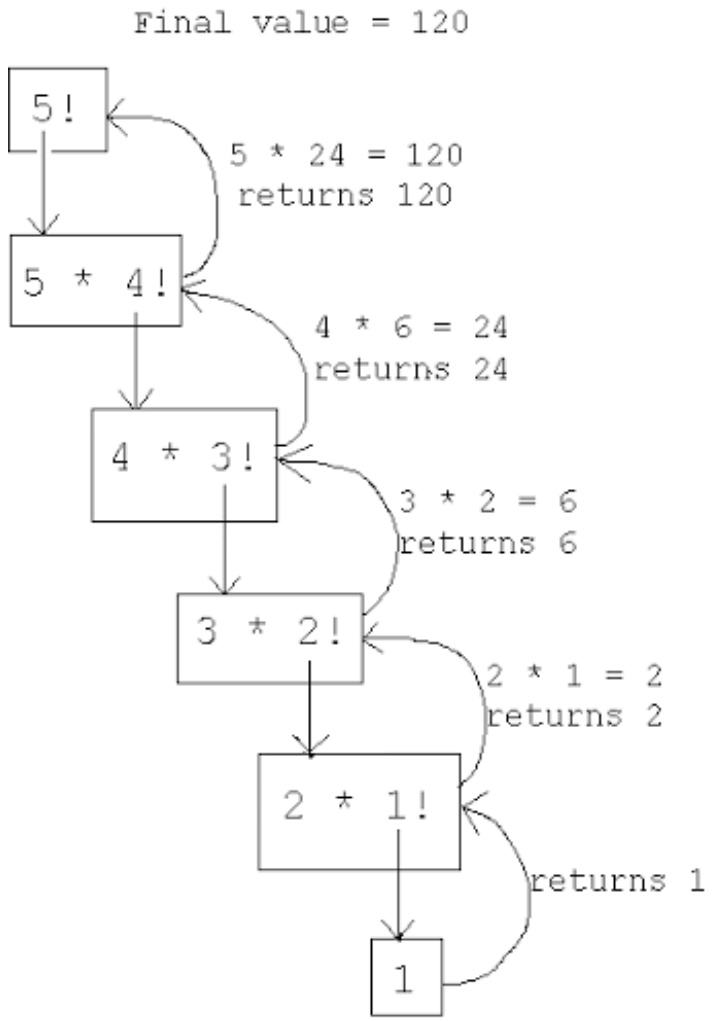
- The solution to the smallest sub problem is called the base case.

- In the recursive program, the solution to thebase case is provided

Data Structures and its Applications

Application of stacks – Recursion

Example 1: Factorial of a Number n



Recursive definition :

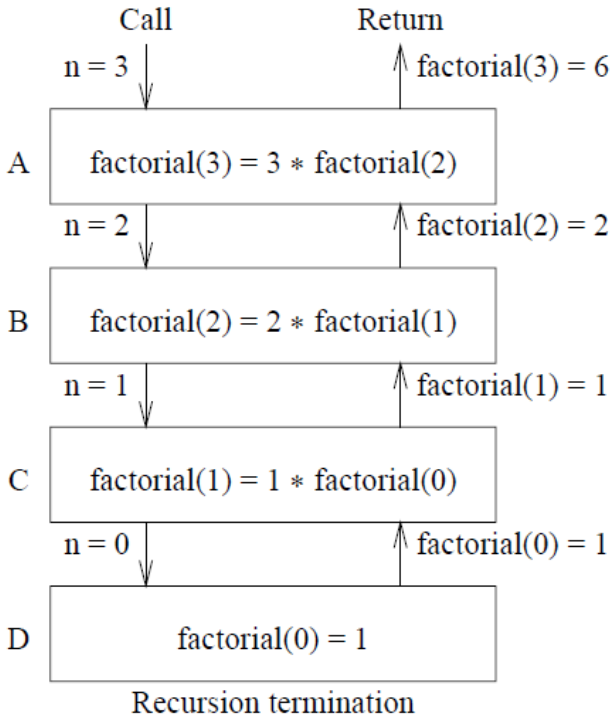
n! = 1 if n=1

n!= n * (n-1)! If n>1

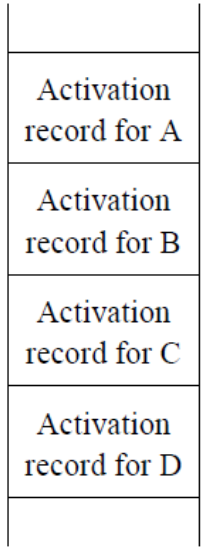
Application of stacks – Recursion

Recursive function to compute factorial of n

```
factorial(n)
{
  int f;
  if(n==0)
       return 1;
   f=n*factorial(n-1);
return f;
}
```



(a)



(b)

Application of stacks – Recursion

Recursive function to find the product of a*b

a*b= a if b=1;

a*b = a*(b-1)+a if b >1;

To evaluate 6 * 3

6*3 = 6*2 + 6 = 6*1 + 6 + 6 = 6 + 6 + 6 = 18

```
multiply(int a, int b)
{
   int p;
    if (b==1)
     return a
   p= multiply(a,b-1) + a;
  return p;
}
```

Application of stacks – Recursion

Fibonacci Sequence

The fibonacci sequence is the sequence of integers
1, 1, 2, 3, 5, 8, 13, 21, 34...

Each element is the sum of two preceding elements

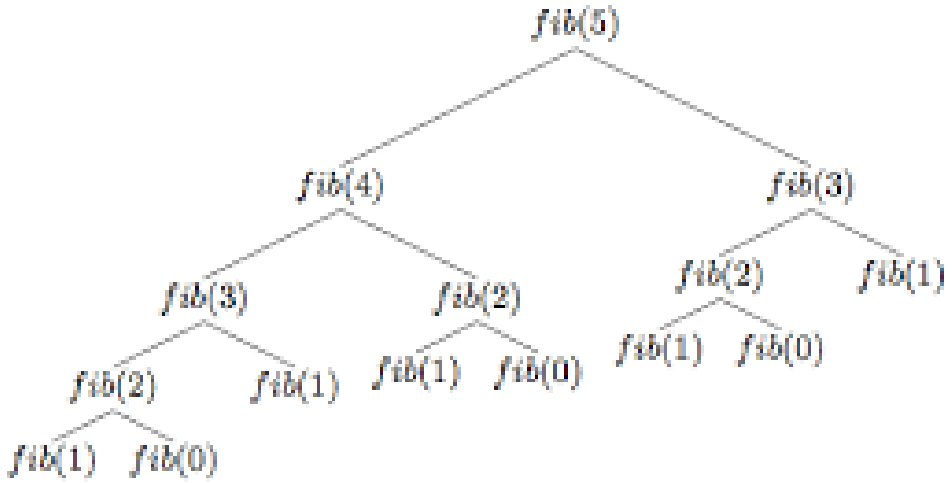
If we consider fib(0) = 1 and fib(1)=1
then recursive definition to compute the nth element in the sequence is

fib(n) = n if n=0 or n=1
fib(n) = fib(n-1) + fib(n-2) for n >=2

Application of stacks – Recursion

Recursive function to compute the nth element in the Fibonacci Sequence

```
 fib(int n)
{
   int x,y;
   if ( (n==0) || (n==1
     return n;
  x= fib(n-1) ;
  y=fib(n-2);
 return x+y;
}
```



Application of stacks – Recursion

Recursive function to find the sum of elements of an array

```
int sum(int *a, int n)
//a is pointer to the array, n is the index of the last element of the array
{
  int s;
  if(n==0) // base condition
   return a[0];
  s= sum(a,n-1) + a[n]; // compute sum of n-1 elements and add the nth element
  return s;
}
```

Application of stacks – Recursion

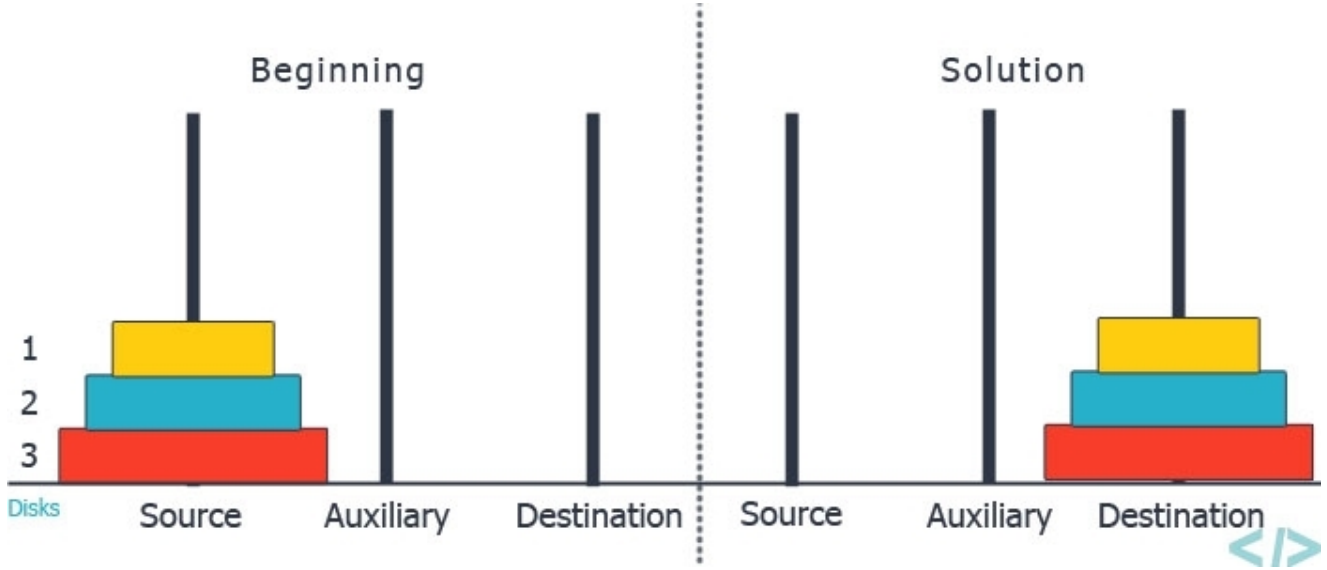
Recursive function to display the elements of the linked list in the reverse order

```
int display(struct node *p)
{
  if(p->next!=NULL)
     display(p->next)
  printf("%d ",p->data);
}
```

Application of stacks – Recursion

Tower of Hanoi

Three Pegs A, B and C exists. N disks of differing diameters are placed on peg A. The Larger disk is always below a smaller disk. The objective is to move the N disks from Peg A to Peg C using Peg B as the auxillary peg



Tower of Hanoi – recursive solution

If a solution to n-1 disks is found, then the problem would be solved. Because in the trivial case of one disk, the solution would be to move the single disk from Peg A to Peg C.

To move n disks from A to C , the recursive solution would be as follows

1. If n=1 move the single disk from A to C and stop
2. Move the top n-1 disks from A to B using C as auxillary
3. Move the remaining disk from A to C
4. Move n-1 disks from B to C using A as the auxillary

Application of stacks – Recursion

Recursive function for Tower of Hanoi

```
void tower(int n,char src,char tmp,char dst)
 {
  if(n==1)
  {
   printf("\nMove disk %d from %c to %c",n,src,dst);
   return;
  }
  tower(n-1,src,dst,tmp);
  printf("\nMove disk %d form %c to %c",n,src,dst);
  tower(n-1,tmp,src,dst);
  return;
 }
```

Application of stacks – Recursion

Recursive function for Tower of Hanoi

Solution for 4 disks

Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 3 from peg A to peg B
Move disk 1 from peg C to peg A
Move disk 2 from peg C to peg B
Move disk 1 from peg A to peg B
Move disk 4 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 2 from peg B to peg A
Move disk 1 from peg C to peg A
Move disk 3 from peg B to peg C
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C

Solution for moving 3 disks from A to B
Move 4^{th} disk from A to C
Solution for moving 3 disks from B to C

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Infix to Postfix and Prefix Expressions – Implementation

Dinesh Singh
Department of Computer Science & Engineering

Infix , Postfix and Prefix Expressions

- Consider the sum of A and B expressed as A + B
- This representation is called infix.
- There are two alternate notations for expressing sum of A and B using the symbols A , B and + , these are
  - Prefix : + A B
  - Postfix : A B +
- The prefixes “Pre” , “post” and “in” refers to the relative position of the operator with respect to the two operands.
- In prefix, operators precedes the two operands
- In postfix, the operator follows the two operands
- In infix, the operator is in between the two operands

Infix , Postfix and Prefix Expressions

Conversion of Infix to Postfix

- For Example : consider the expression A + B * C
- The evaluation of the above expression requires the knowledge of precedence of operators
- A + B * C can be expressed as A + (B * C) as multiplication takes precedence over addition
- Applying the rules of precedence the above infix expression can be converted to postfix as follows

A + B * C = A + (B * C)

= A + (BC *) convert the multiplication

= A (B C *) + convert the addition

= A B C * +

Conversion of Infix to Prefix

- For Example : consider the expression A + B * C
- Applying the rules of precedence the above infix expression can be converted to prefix as follows

A + B * C = A + (B * C)

= A + (* B C) convert the multiplication

= + A (* B C) + convert the addition

= + A * B C

Infix , Postfix and Prefix Expressions

Applying the rules of precedence the table shows the conversion of Infix to Postfix and Prefix Expression

Infix	Postfix	Prefix
A + B * C + D	A B C * + D +	+ + A * B C D
(A + B) * (C + D)	A B + C D + *	* + A B + C D
A * B + C * D	A B * C D * +	+ * A B * C D
A + B + C + D	A B + C + D +	+ + + A B C D
A $ B * C – D + E / F / (G+H)	A B $ C * D – E F / G H + / +	+ - * $ A B C D / /E F + G H
((A + B)*C – (D – E)) $ (F + G)	A B + C * D E – - F G + $	$ - * + A B C – D E + F G
A – B / (C * D $ E)	A B C D E $ * / -	- A / B * C $ D E

$ is the exponentiation operator and its precedence is from right to left

For example A $ B $ C = A $ (B $ C)

Infix to postfix conversion - algorithm

```
opstk is the empty stack
while(not end of input)
{
  symb = next input character
// if the input symbol is an operand , add it to the postfix string
 if(symb is an operand)
    add symb to postfix string
else
   {
     // pop the contents of the stack while the precedence of
    // top of the stack is greater than the precedence of the symbol scanned
    //prcd function returns true in this case
      while(!empty(opstk ) and ( prcd(stacktop(opstk),symb))
       {
          topsymb=pop(opstk)
         add topsymb to postfix string
       }
```

Infix to postfix conversion - algorithm

```
   /* if prcd function returns false, then
if  the stack is empty and symbol is not ‘)’, push symb on to the stack*/
        if( empty(opstk) || symb!=‘)’)
            push(opstk,symb)
      else
          topsymb=pop(opstk); // if symb is ‘)’ ,  pop ‘ ( ‘ from the stack
}
while(!empty(opstk)
 {
   topsymb=pop(opstk)
   add topsymb to postfix string
 }
}
```

Infix to postfix conversion - algorithm

- prcd(OP1,OP2) is a function that compares the precedence of the top of the stack(OP1) and the input symbol (OP2) and returns TRUE if the precedence is greater else returns FALSE.
- Some of the return values of prcd function
  - prcd(‘ * ‘ , ‘ + ‘) returns TRUE : prcd(‘ * ’, ’ / ’) returns TRUE
  - prcd(‘ + ‘, ‘ * ‘) returns FALSE : prcd(‘ / ’, ’ * ’) returns TRUE
  - prcd(‘ + ‘ , ‘ + ‘) returns TRUE :
  - prcd(‘ + ‘ , ‘ – ‘) returns TRUE
  - prcd(‘ – ‘ , ‘ – ‘) returns TRUE
  - prcd(‘ $ ‘ , ‘$’) returns FALSE : exponentiation operator, associatively from right to left
  - prcd(‘ (‘, op) , prcd(op ‘ (‘) returns FALSE for any operator
  - prcd (op , ‘)‘) returns TRUE for any operator
  - Prcd(‘ (‘ , ‘) ‘) returns FALSE

Infix to postfix conversion - algorithm

- The motivation behind the conversion algorithm is the desire to output the operators in the order in which they are to be executed.
- If an incoming operator is of greater precedence than the one on top of the stack, this new operator is pushed on to the stack.
- If on the other hand , the precedence of the new operator is less than the one on the top of the stack, the operator on the top of the stack should be executed first.
- Therefore the top of the stack is popped out and added to the postfix and the incoming symbol is compared with the new top and so on.
- Parenthesis in the input string override the order of operations
- When a left parenthesis is scanned, it is pushed on to the stack
- When its associated right parenthesis is found, all the operators between the two parenthesis are placed on the output string. Because they are to be executed before any operators appearing after the parenthesis

Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for A + B * C

symb	postfix string	opstk	Remarks
A	A	Empty	symb is operand , add 'A'on to the postfix string
+	A	+	symb is operator, and stack empty, push + on to the stack
B	AB	+	symb is operand , Add 'B' to the postfix string
*	AB	+*	symb is operator, precedence of + (stack top) is < than precedence of *, therefore push * on to the stack

Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for A + B * C

symb	postfix string	opstk	Remarks
C	ABC	+ *	symb is operand , add C on to the postfix string
End of input	ABC*	+	Pop * from the stack and add to the postfix string
-	ABC*+	Empty	Pop + from the stack and add to the postfix string

Data Structures and its Applications

Infix to postfix conversion – Trace of the algorithm

PES UNIVERSITY ONLINE

Trace of the algorithm for (A + B)* C

symb	postfix string	opstk	Remarks
(	-	Empty	Stack empty, push '(' on to the stack
A	A	(	Symb is an operand, add 'A' to thepost fix string
+	A	(+	Precedence of top of the stack '(' is less than '+' , push '+' on to the stack
B	AB	(+	Symb is an operand, add B to the postfix string
)	AB+	(	Precedence of stack top '+' is > than ')' Pop '+' from the stack and add to the postfix string
	AB+	empty	Precedence of stack top stack top '(' is not greater than ')' and symb is ')', therefore pop '(' from the stack

Infix to postfix conversion – Trace of the algorithm

Trace of the algorithm for (A + B)* C

symb	postfix string	opstk	Remarks
*	AB+	*	Stack empty, push '*' on to the stack
C	AB+C	*	Symb is an operand, add 'C' to the postfix string
End of string	AB+C*	Empty	End of input, pop * from the stack and add to the postfix.

Infix to posfix conversion – another algorithm

```
void convert_postfix(char *infix,char*postfix)
 {
    i=0;
   char ch;
   j=0;
   push(s,&top,'#');
   while(infix[i]!='\0')
   {
     ch=infix[i];
   //while the prcedence of top of stack is greater than the
//precedence of the input symbol, pop and add to the postfix
    while(stack_prec(peep(s,top))>input_prec(ch))
        postfix[j++]=pop(s,&top);
```

Infix to posfix conversion – another algorithm

```
if(input_prec(ch)!=stack_prec(peep(s,top)))
        push(s,&top,ch);
      else
        pop(s,&top);
    i++;
   }
   while(peep(s,top)!='#')
     postfix[j++]=pop(s,&top);
   postfix[j]='\0';
 }
```

Infix to posfix conversion – another algorithm

Precedence table

Operator	Input Precedence	Stack Precedence
+ , -	1	2
* , /	3	4
$	6	5
Operands	7	8
)	0	- : Never pushed on to stack
(	9	0
#	-	-1

Infix to prefix conversion – algorithm

Example 1:
Input : A * B + C / D
Output : + * A B/ C D

Example 2 :
Input : (A - B/C) * (D/K-L)
Output : *-A/BC-/DKL

1: Reverse the infix expression i.e A+B*C will become C*B+A.
Note while reversing each '(' will become ')' and each ')' becomes '('.

2: Obtain the postfix expression of the modified expression i.e CB*A+.

3: Reverse the postfix expression. Hence in our example prefix is +A*BC.

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Evaluation of Postfix expression and Parenthesis matching

Dinesh Singh
Department of Computer Science & Engineering

Evaluation of Postfix Expression - Algorithm

- Each operator in a postfix string refers to the previous two operands.
- Each time an operand is read , it is pushed on to the stack
- When an operator is reached, its operands will be the top two elements on to the stack.
- The two elements are popped out, the indicated operation is performed on them and result is pushed on the stack so that it will be available for use as an operand of the next operator.

Evaluation of Postfix Expression - Algorithm

```
opndstk is the empty stack
while(not end of the input) // scan the input string
{
 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 opn2;
   push(opndstk, value);
  }
 return(pop(opndstk));
}
```

Evaluation of Postfix Expression – Trace of the Algorithm

Infix : 3 + 5 * 4

Postfix expression : 3 5 4 * +

Symb	Opnd1	Opnd2	Value	opndstk
3	-			3
5				3, 5
4				3 , 5 , 4
*	5	4	20	3 , 20
+	3	20	60	60

Data Structures and its Applications

Evaluation of Postfix Expression - implementation

```
int postfix_eval(char* postfix)
 {
  int i,top,r;
  int s[10];//stack
  top=-1;
  i=0;

  while(postfix[i]!='\0')
  {
   char ch=postfix[i];
   if(isoper(ch))
   {
    int op1=pop(s,&top);
    int op2=pop(s,&top);
```

Evaluation of Postfix Expression - implementation

```
switch(ch)
      {
        case '+':r=op1+op2;
                 push(s,&top,r);
                 break;
        case '-':r=op2-op1;
                  push(s,&top,r);
                  break;
         case '*':r=op1*op2;
                 push(s,&top,r);
                  break;
        case '/':r=op2/op1;
                  push(s,&top,r);
                 break;
       }//end switch
    }//end if
```

Evaluation of Postfix Expression - implementation

```
else
      push(s,&top,ch-'0');//convert charcter to
integer and push
     i++;
    }  //end while
  return(pop(s,&top));
 }
```

Parenthesis Matching – overview of the Algorithm

Examples

1. (( )) : Valid Expression
2. ((( )) : Invalid Expression (Extra opening parenthesis)
3. (( ))) : Invalid Expression (Extra closing parenthesis)
4. ((}) : Invalid Expression (Parenthesis mismatch)
5. (( )] : Invalid Expression (Parenthesis mismatch)

Parenthesis Matching – overview of the Algorithm

1.Read the input symbol from the input expression
2. If the input symbol is one of the open parenthesis (‘(‘ , ‘ { ‘ or ‘ [‘), it is pushed on to the stack
3. If the input symbol is of closing parenthesis, stack is popped and the popped parenthesis is compared with the input symbol, if there is a mismatch in the type of the parenthesis, return 0 (Mismatch of parenthesis)
4. If there is a match in the parenthesis , the next input symbol is read.
5. If during this process, if the stack becomes empty, return 0 (Extra closing parenthesis)
6. If at the end of the expression, if the stack is not empty, return 0 (Extra opening parenthesis)
7. If at the end of the input expression, if the stack is empty, return 1. (Parenthesis are matching)

Data Structures and its Applications

Parenthesis Matching - Implementation

```
int match(char *expr)
 {
   int i,top;
   char s[10],ch,x;//stack
   i=0;
    top=-1;

   while(expr[i]!='\0')
   {
     ch=expr[i];
     switch(ch)
     {
        case '(':
        case '{':
        case '[':push(s,&top,ch);
                 break;
```

Parenthesis Matching

```
case ')':if(!isempty(top))
                {
                  x=pop(s,&top);
                  if(x=='(')
                    break;
                  else
                    return 0;//mismatch of parenthesis
                }
                else
                  return 0;//extra closing parenthesis
```

Parenthesis Matching

```
case '}':if(!isempty(top))
         {
          x=pop(s,&top);
          if(x=='{')
            break;
          else
            return 0;//mismatch of parenthesis
         }
         else
          return 0;//extra closing parenthesis
```

Data Structures and its Applications

Parenthesis Matching

```
case ']':if(!isempty(top))
                {
                 x=pop(s,&top);
                 if(x=='[')
                   break;
                 else
                   return 0;//mismatch of parenthesis
                }
                else
                 return 0;//extra closing parenthesis

        }//end switch
        i++;
    }//end while
   if(isempty(top))
      return 1;
    return 0;//extra opening parenthesis
  }
```

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

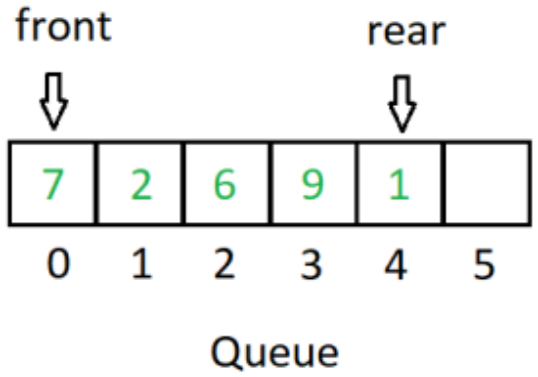
DATA STRUCTURES AND ITS APPLICATIONS

Basic Structure of a Queue , Implementation using an Array

Dinesh Singh
Department of Computer Science & Engineering

Queue Data Structure - definition

- A Queue is an ordered collection of items from which items may be deleted at one end (called the front of the queue)and into which items may be inserted at the other end (called the rear of the queue).



Types Queue

- Different Types of Queues
  - Simple Queue
  - Circular Queue
  - Priority Queue
  - Dequeue

- Implementation
  - Sequential Representation (Arrays)
  - Linked Representation (Linked Lists)

Queue Operations

- Three primitive operations can be applied to the queue
- Insert (q,x) : inserts x at the rear of the queue q
- x = remove (q) : deletes front element from the queue and set x to its contents
- empty(q) : returns true or false depending on whether the queue contains any elements.
- Insert operation cannot be performed if the queue has reached the maximum size.
- The result of an illegal attempt to insert an element into a queue which has reached its maximum size is called overflow.
- The remove operation can be applied only if the queue is non empty.
- The result of an illegal attempt to remove an element from an empty queue is called underflow.
- The empty operation is always applicable.

Structure of a Simple Queue – Sequential Representation

```
#define MAXQUEUE 100
struct queue
{
  int items [MAXQUEUE];
  int front, rear;
} ;

struct queue q;
q.rear = q.front = -1;
```

Functions to implement the operations

- insert (x, &q)
- remove (&q)
- empty(&q)

Simple Queue – Insert Operation

Inserting into a queue

1. Check the queue for overflow condition
2. If the queue is not in overflow condition, increment the rear pointer and insert the element at a location indicated by the rear pointer.
3. If this is the first element in the queue, initialise front to 0.
4. Return 1

Simple Queue – Implementation of Insert

```
int insert(int x,struct queue *q)
 {
  //check queue overflow
 if(q->rear==MAXQUEUE - 1)
 {
  printf("Queue overflow..\n");
  return -1;
 }
 (q->rear)++;
 q->items[q->rear]=x;
 if(q->front==-1)//if first element
   q->front=0;
 return 1;
}
```

Simple Queue – Remove Operation

Removing element from the queue

1. Check the queue for underflow condition
2. If the queue is not in underflow condition, remove the element pointed by the front pointer into x.
3. If this was the only element in the queue, initialise front and rear to -1.
4. Return x

Simple Queue – Implementation of remove

```
int remove(struct queue *q)
 {
  int x;

  if(q->front==-1)
  {
    printf("Queue empty..\n");//underflow
    return -1;
  }
   x=q->items[q->front];
   if(q->front==q->rear)//only one element in queue
    q->front=q->rear=-1;
   else
    (q->front)++;
   return x;
 }
```

Simple Queue – Implementation of empty and display

```
int empty ( struct queue * q)
{
  if (q->front ==-1)
    return 1;
  return -1;
}
display (struct queue * q)
{
 int i;
 if(q->front==-1)
   printf(”Empty queue..\n”)
 else
 {
   for ( i = q->front; i<=q->rear;i++)
     printf(”%d”,q->items[i]);
 }
}
```

Simple Queue – Another Implementation

Implementation where queue represented as an array , front and rear are separate variables

```
#define MAXQUEUE 100

int q[MAXQUEUE]
int front, rear
front = rear = -1
```

Function calls to implement the operations

- insert(x, q, &front, &rear) ;
- remove (q , &front, &rear);
- empty(q, &front)

Simple Queue – Another Implementation

Implementation where queue represented as an array , front and rear are separate variables

```
int insert(int x,int *q,int *f,int *r)
 {
  //check queue overflow
 if(*r==MAXQUEUE - 1)
 {
  printf("Queue overflow..\n");
  return -1;
 }
 (*r)++;
 q[*r]=x;
 if(*f==-1)//if first element
   *f=0;
 return 1;
}
```

Simple Queue – Another Implementation

Implementation where queue represented as an array , front and rear are separate variables

```
int remove(int *q,int *f,int *r)
 {
  int x;
  if(*f==-1)
  {
    printf("Queue empty..\n");
    return -1;
  }
   x=q[*f];
   if(*f==*r)//only one element in queue
     *f=*r=-1;
   else
     (*f)++;
   return x;
 }
```

Data Structures and its Applications

Simple Queue – Another Implementation

Implementation where queue represented as an array , front and rear are separate variables

```
display (int *q, int f, int r)
{
  int i;
  if(f==-1)
    printf(”Empty queue..\n”)
  else
  {
    for ( i = f; i<= r; i++)
      printf(”%d”,q[i]);
  }
}
```

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Circular Queue - Implementation

Dinesh Singh
Department of Computer Science & Engineering

Circular Queues - definition

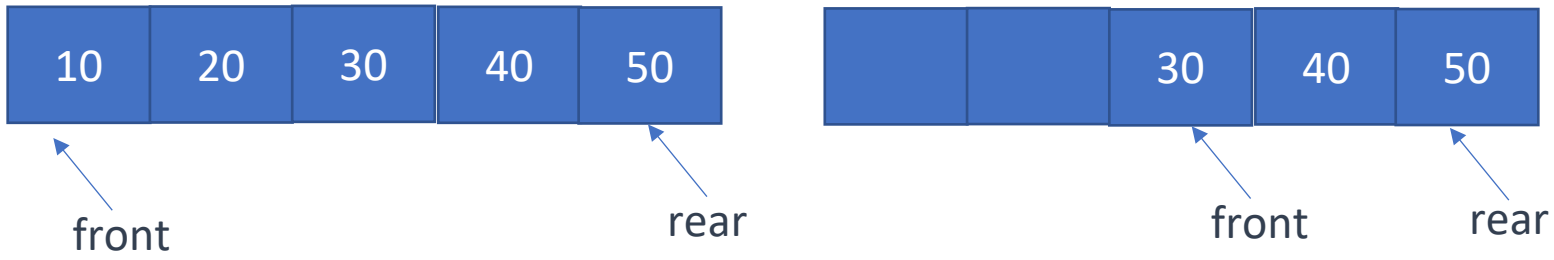
- Circular Queue is a linear data structure, which follows the principle of FIFO(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

- In a simple queue, once the queue is completely full, it's not possible to insert more elements. Even if we perform remove operation on the queue to remove some of the elements, until the queue is reset, no new elements can be inserted

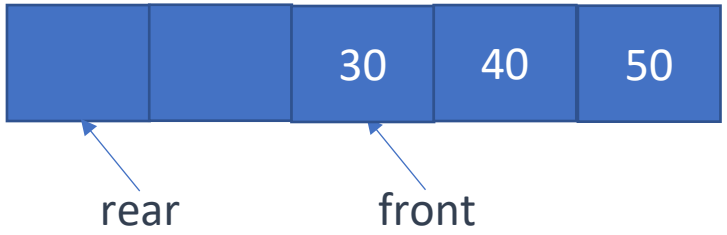
Data Structures and its Applications

Drawback of a simple Queue

Structure of the simple queue



Cannot insert even after two elements are removed and Space available in the front.



It is possible to insert in a circular queue by moving the rear To the beginning of the queue

Implementation of operations

- To insert into the queue : Finding the rear index

```
rear = (rear + 1) % size
If(rear=front)
   cannot insert
else
   insert at rear index
```

- For eg. If size = 5, front = 2 and rear = 4
- rear = (4 + 1) % 5 = 0,
- The new element gets inserted at index 0

- For eg. If size = 5, front = 0 and rear = 4
- rear = (4 + 1) % 5 = 0,
- rear = front , therefore cannot insert

Implementation of operations

- To remove from the queue :

  remove the element pointed by front , move the front

  front = (front + 1) % size

For eg. If size = 5, front = 2 and rear = 4

- front = (2 + 1) % 5 = 3,
- front moves to index 3 after removal of the element,

- For eg. If size = 5, front = 4 and rear = 2
- front = (4 + 1) % 5 = 0,
- Front moves to 0 after removal of the element

Structure of a circular Queue – Sequential Representation

```
#define MAXQUEUE 100
struct queue
{
  int items [MAXQUEUE];
  int front, rear;
} ;

struct queue q;
q.rear = q.front = -1;
```

Functions to implement the operations

- insert (&q,x)
- remove (&q)

Data Structures and its Applications

Implementation of operations - insert

```
int qinsert(struct queue *q, int x)
{

 //check for queue overflow
      if((q->r+1)%MAXQUEUE==q->f)
      {
           printf("Queue Overflow..\n");
            return -1;
       }
      else
       {
            q->rear=(q->rear+1)%size;     //get the rear index
            q->item[q->rear]=x;    //insert at rear index
            if(q->front==-1)  //if first element
              q->front=0;     // make front point to 0
            return 1;
             }
    }
```

Data Structures and its Applications

Implementation of operations - insert

```
//ANOTHER WAY TO IMPLEMENT INSERT
int qinsert(int *q,int *f, int *r, int size, int x)
{
 // f ,r are pointers to front and rear of the queue, size is the max size of queue
 //check for queue overflow
      if((*r+1)%size==*f)
      {
           printf("Queue Overflow..\n");
            return -1;
       }
      else
       {
            *r=(*r+1)%size;     //get the rear index
            q[*r]=x;    //insert at rear index
            if(*f==-1)  //if first element
              *f=0;     // make front point to 0
            return 1;
             }
   }
```

Data Structures and its Applications

Implementation of operations - remove

```
int remove(struct queue *q)
 {
    int x;
   if(q->front==-1)  //check for empty queue
  {
    printf("Queue empty..\n");
    return -1;
  }
else
   {
      x=q->items[q->front];
      if(q->front==q->rear)//only one element
         q->front=q->rear=-1;
 else
        q->frontf=(q->front+1)%MAXQUEUE;   //increment the front
      return x;
    }
}
```

Implementation of operations - remove

```
//ANOTHER WAY TO IMPLEMENT REMOVE
int remove(int *q, int *f, int *r,int size)
 {
   int x;
  if(*f==-1)  //check for empty queue
  {
   printf("Queue empty..\n");
   return -1;
  }
else
  {
    x=q[*f];
    if(*f==*r)//only one element
      *f=*r=-1;
 else
      *f=(*f+1)%size;   //increment the front
    return x;
   }
}
```

Implementation of operations - display

```
void display(struct queue q)
 {
  if(q.front==-1)
    printf("\nQueue empty..\n");
  else
  {
    while(q.front!=q.rear) //increment front till it reaches rear
    {
      printf("%d ",q.items[q.front]);
      q.front=(q.front+1)%MAXQUEUE;
    }
   printf("%d ",q->items[q->front]); // display the last element
  }
}
```

Implementation of operations - display


```
void display(int *q, int f, int r, int size)
 {
  if(f==-1)
    printf("\nQueue empty..\n");
  else
  {
    while(f!=r) //increment front till it reaches rear
    {
      printf("%d ",q[f]);
      f=(f+1)%size;
    }
   printf("%d ",q[f]); // display the last element
  }
}
```

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Priority Queue - Implementation

Dinesh Singh
Department of Computer Science & Engineering

Priority Queues - definition

- Priority Queue is an extension of queue where every item has a priority associated with it.
- There are two types of priority queue
  - Ascending priority queue
  - Descending Priority queue
- In Ascending Priority queue ,the item with lowest priority is removed
- In descending priority queue, the item with the highest priority is removed.

Priority queues - Operations

- Operations in Ascending priority queue
  - pqinsert(apq,x) : inserts element x into the priority queue apq.
  - pqmindelete(apq) : removes the element with the minimum priority.
- Operations in Descending priority queue
  - pqinsert(dpq,x) : inserts element x into the priority queue dpq.
  - pqmaxdelete(dpq) : removes the element with the maximum priority.
- The operation empty(pq) determines whether the queue is empty or not.

Priority Queues- alternate definition

- In alternate definition of a priority queue, the items do not have a priority but the intrinsic ordering of elements determine results of its basic operation.
- In ascending priority queue, items can be inserted arbitrarily, but only the smallest element can be removed.
- In descending priority queue, items can be inserted arbitrarily, but only the largest element can be removed.

Priority Queues- Implementation

- Priority queues can be implemented by
  - Arrays
  - Linked Lists
  - Heap

Priority Queues- Array Implementation

```
struct pqueue
{
  int  data;
  int pty;
}
struct pqueue pq[10];
```

implementation of descending priority :

Items inserted based on their priority

- Insert : The item is inserted based on its priority. The queue is ordered in the descending priority of the items. The item with the highest priority will be at the first location in the array.

- Remove : delete always the first element, Shift the other elements to the left after the deletion

Priority Queues- Array Implementation

implementation of descending priority :

Items inserted arbitrarily

- Insert : The item is inserted at the rear of the queue. The queue is therefore unordered.
- Delete : The item with the highest priority is found and removed. The items following the position of the removed element are shifted to the left.

Priority Queues- Array Implementation

implementation of descending priority where items are inserted according to the priority

```
void pqinsert(int x,int pty,struct pqueue *pq,int *count)
{
 // x is item to be inserted
 // pty is the priority of the item
// pq is the pointer to the priority queue
// count is the number of items in the queue

    int j;
    struct pqueue key;
    key. data=x;
    key.pty=pty;
```

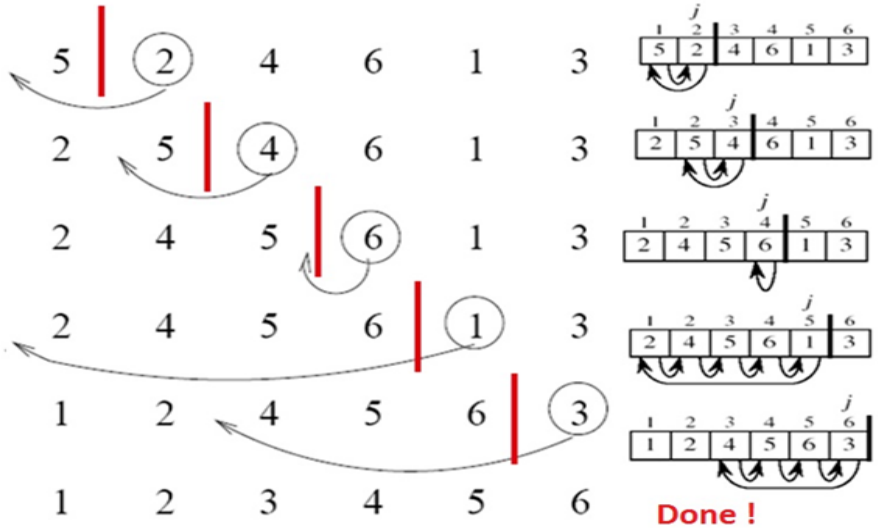
Priority Queues- Array Implementation

implementation of descending priority where items are inserted according to the priority

```
j=*count-1; // index of the initial position of the element

//compare the priority of the item being inserted with the
//priority of the items in the queue
// shift the items down while the priority of the item being
//inserted is greater than priority of the item in the queue

while((j>=0)&&(pq[j].pty>key.pty))
     {
          pq[j+1]=pq[j];
           j--;
      }
      pq[j+1]=key; // insert the element at its correct location
      (*count)++;
 }
```



Priority Queues- Array Implementation

implementation of descending priority where items are inserted according to the priority

```
struct pqueue pqdelete(struct pqueue *pq,int *count)
{
    // pq is a pointer to the priority queue
    // count is the number of elements in the queue

     int i;
     struct pqueue key;
     // if queue is empty, return a structure with priority -1

     if(*count==0)
       {
           key.data=0;
           key.pty=-1;
       }
```

Priority Queues- Array Implementation

implementation of ascending priority where items are inserted according to the priority

```
    //delete the first item
    //shift the other items to the left
     else
     {
          key=pq[0];
          for(i=1;i<=*count-1;i++)
               pq[i-1]=pq[i];
          (*count)--;
     }
     return key;  //return the key with the lowest priority
}
```

Priority Queues- Array Implementation

PES UNIVERSITY ONLINE

Implement the following

- Ascending Priority queue where item is inserted at the end and item with the lowest priority is deleted.
- Descending priority queue
- Ascending and Descending Priority queue using a linked list

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Dequeue - Implementation

Dinesh Singh
Department of Computer Science & Engineering

Dequeue(Double ended Queue) - definition

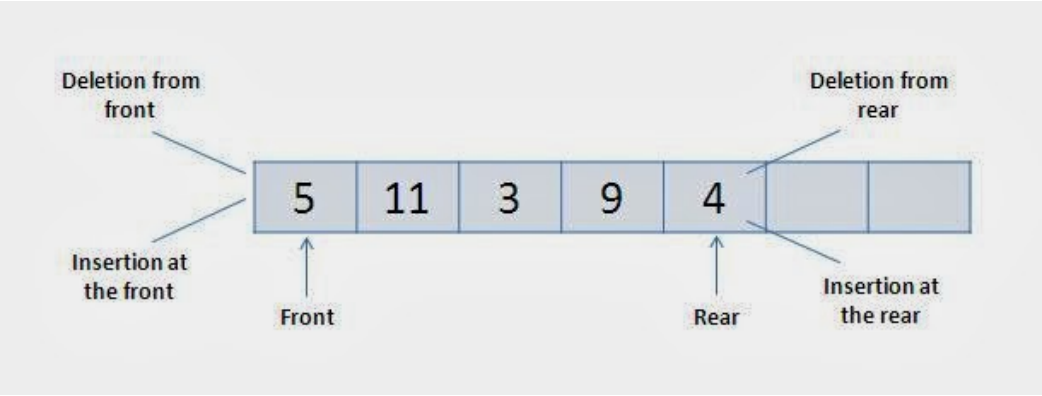
Double ended queue is a queue that allows insertion and deletion at both ends.

Data Structures and its Applications

Dequeue(Double ended Queue) - definition

The following four basic operations are performed on dequeue:

- *insertFront()*: Adds an item at the front of Deque.
- *insertRear()*: Adds an item at the rear of Deque.
- *deleteFront()*: Deletes an item from front of Deque.
- *deleteRear()*: Deletes an item from rear of Deque.



Data Structures and its Applications

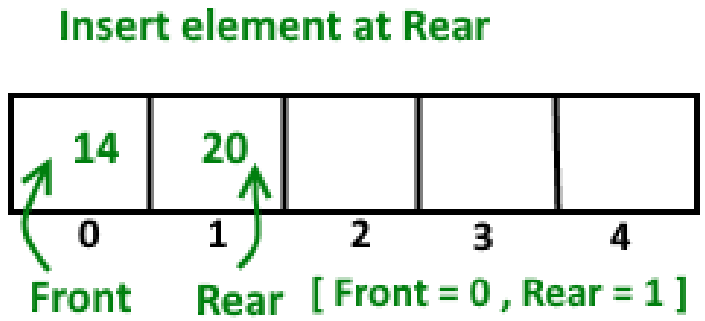
Dequeue(Double ended Queue) - Array Implementation

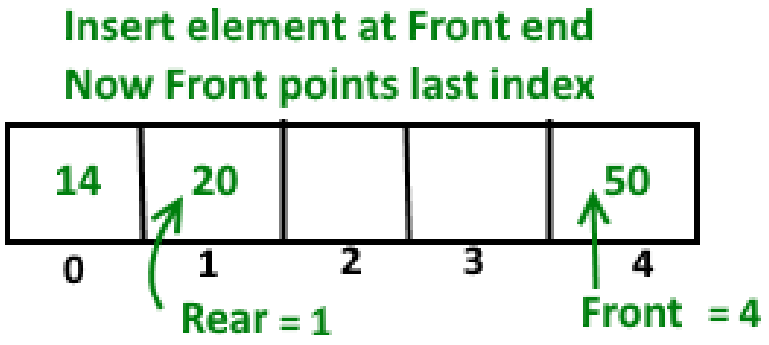
Insert Elements at Rear end :

Check whether the queue is full
If rear = size-1
 initialise rear to 0.
else
  increment rear by 1
 insert element at location rear

Insert element front end

Check if the queue is full
If Front =0
 move front to last location (size -1)
else
 decrement front by 1
 insert at location front





Data Structures and its Applications

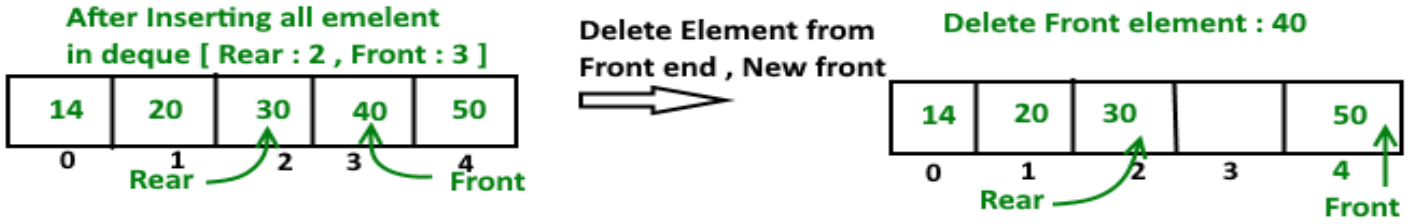
Dequeue(Double ended Queue) - Array Implementation

Delete element at Rear end

check if the queue is empty
delete the element pointed by rear
If dequeue has one element
  front=-1 rear=-1;
If rear is at first index
  make rear = size-1
else
  decrease rear by 1

Delete element at front end

check if the queue is empty
delete the element pointed by front
If dequeue has one element
  front=-1 rear=-1;
If front is at last index
  make front = 0
else
  increase front by 1



Dequeue(Double ended Queue) - Doubly Linked list Implementation

Structure of Dequeue

```
struct dequeue
{
   struct node * front;
   struct node * rear;
};
struct node
{
    int data;
    struct node * prev, *next;
};
struct   dequeue dq;

dq.front=dq.rear = NULL
```

Data Structures and its Applications

Dequeue(Double ended Queue) - - Doubly Linked list Implementation

```
//insert in front of the queue
void qinsert_head(int x,struct dequeue *dq)
  {
    struct node *temp;

    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
    temp->prev=temp->next=NULL;

    if(dq->front==NULL) // first element
      dq->front=dq->rear=temp;
     else
      {
        temp->next=dq->front;  // insert in front
        dq->front->prev=temp;
        temp->prev=NULL;
        dq->front=temp;
      }
  }
```

Data Structures and its Applications

Dequeue(Double ended Queue) - - Doubly Linked list Implementation

```
//insert at the rear of the queue
void qinsert_tail(int x,struct dequeue* dq)
 {
  struct node *temp;

   temp=(struct node*)malloc(sizeof(struct node));
   temp->data=x;
   temp->prev=temp->next=NULL;

   if(dq->front==NULL)
     dq->front=dq->rear=temp;
    else
     {
       dq->rear->next=temp;
       temp->prev=dq->rear;
       dq->rear=temp;
     }
 }
```

Dequeue(Double ended Queue) - - Doubly Linked list Implementation

```
//delete at the front of the queue
int qdelete_head(struct dequeue* dq)
 {
   struct node *q;
   int x;
   if(dq->front==NULL)
     return -1;

  q=dq->front;
   x=q->data;
  if(dq->front==dq->rear)//only one node
    dq->front=dq->rear=NULL;
  else
   {
     dq->front=dq->front->next;
     dq->front->prev=NULL;
   }
   free(q);
  return x;
 }
```

Data Structures and its Applications

Dequeue(Double ended Queue) - - Doubly Linked list Implementation

```
//delete at the rear of the queue
int  qdelete_tail(struct dequeue* dq)
 {
  struct node *q;
   int x;
   if(dq->front==NULL)
     return -1;
   q=dq->rear;
   x=q->data;
  if(dq->front==dq->rear)//only one node
    dq->front=dq->rear=NULL;
  else
   {
     dq->rear=dq->rear->prev;
     dq->rear->next=NULL;
    }
   free(q);
    return x;
 }
```

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

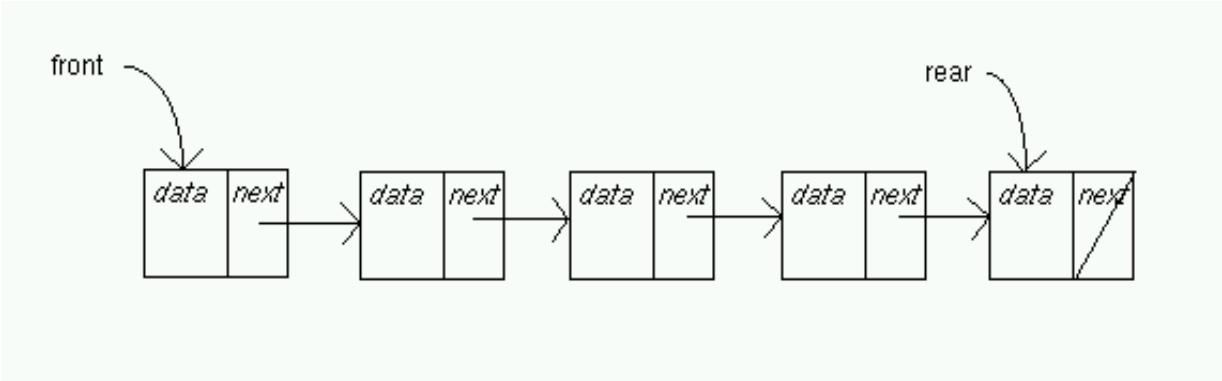
Queues – Linked List Implementation

Dinesh Singh
Department of Computer Science & Engineering

Queues - Linked list Implementation

In a linked list implementation two pointers are maintained : front and rear .

- front points to the first item of the queue
- rear points to the last item of the queue



Queues - Linked list Implementation

Operations :

- Insert() : adds a new node after the rear and moves rear to the next node
- Remove() : removes the first node and moves front to the next node
- Empty() : Checks if the queue is empty

Queues - Linked list Implementation

Structure of queue

```
struct node
{
  int data;
  struct node *next;
};
struct queue
{
   struct node * front;
   struct node *rear;
};

Struct queue q;
q.front=q.rear = NULL;
```

Queues - Linked list Implementation – Operations

Insert operation

```
Insert(q,x)

p=getnode();
initialise the node
if(q.rear=NULL)
    q.front=p;
else
   next(q.rear) =p;
q.rear = p;
```

remove operation

```
remove(q)

If(empty(q)
   print empty queue
else
  p=q.front;
  x=info(p);
  q.front = next(p);
  if(q.front =NULL)
     q,rear=NULL
  freenode(p);
  return x;
```

Data Structures and its Applications

Queues - Linked list Implementation – Operations

Insert operation of queue implemented by a linked list

```
void qinsert(struct node * q, int x)
  {
    struct node *temp;

    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
    temp->next=NULL;

     //if this is the first node
    if(q->front==NULL)
      q->front=q->rear=temp;
    else //insert at the end
      {
         q->rear->next=temp;
        q->rear=temp;
       }
   }
```

Queues - Linked list Implementation – Operations

remove operation of a queue implemented by a linked list

```
int qremove(struct queue * q)
  {
      struct node *p;
      int x;
      p=q->front;
      if(p==NULL)
       {
        printf("Empty queue\n");
        return -1;
       }
```

Queues - Linked list Implementation – Operations

```
else
        {
         x=q->data;
         if(q->front==q->rear) //only one node
           q->front=q->rear=NULL;
         else
         {
         q->front=q->next; // move front to next node
         return x;
         }
        free(q);
       }
  }
```

Queues - Linked list Implementation – Operations

```
void qdisplay(struct queue q)
    {
      struct node * f, *r;
      if(q.front==NULL)
        printf("Queue Empty\n");
     else
      {
      f=q.front; r=q.rear;
      while(f!=r)
      {
        printf("%d-> ",f->data);
        f=f->next;
      }
     printf("%d-> ",f->data); // print the last node
     }
   }
```

Queues - Linked list Implementation – Operations

Insert operation in an alternate way

```
void qinsert(int x, struct node **f, struct node **r)
//f and r are pointers to variables front and rear of a queue
   {
    struct node *temp;

    temp=(struct node*)malloc(sizeof(struct node));
    temp->data=x;
    temp->next=NULL;

     //if this is the first node
    if(*f==NULL)
      *f=*r=temp;
    else //insert at the end
      {
       (*r)->next=temp;
        *r=temp;
      }
  }
```

Queues - Linked list Implementation – Operations

Remove operation in an alternate way

```
int qdelete(struct node **f, struct node **r)
  {
      struct node *q;
      int x;
      q=*f;
      if(q==NULL)
       {
        printf("Empty queue\n");
        return -1;
       }
else
      {
       x=q->data;
       if(*f==*r) //only one node
         *f=*r=NULL;
       else
       {
       *f=q->next;
       return x;
       }
      free(q);
     }
}
```

Queues - Linked list Implementation – Operations

Disadvantages of representing queue by a linked list

- A node in linked list occupies more storage than the corresponding element in an array.
- Two pieces of information per element is necessary in a list node, where as only one piece of information is needed in an array implementation

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Queues – Implementation of CPU scheduler using Queue

Dinesh Singh
Department of Computer Science & Engineering

Queues – Implementation of CPU scheduler using queue

Different Scheduling Algorithms:

First Come First Serve CPU Scheduling:

- Simplest scheduling algorithm that schedules according to arrival times of processes.
- First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU.
- It is implemented by using the simple queue. When a process enters the ready queue, its PCB is linked onto the rear of the queue.
- When the CPU is free, it is allocated to the process at the front of the queue.
- The running process is then removed from the queue.

Queues – Implementation of CPU scheduler using queue

Shortest Job First(Preemptive):

- In Preemptive Shortest Job First Scheduling, jobs are put into the ready queue as they arrive
- As a process with short burst time arrives, the existing process is preempted or removed from execution, and the shorter job is executed first

Shortest Job First(Non-Preemptive):

- In Non-Preemptive Shortest Job First, a process which has the shortest burst time is scheduled first.
- If two processes have the same bust time then FCFS is used to break the tie

Queues – Implementation of CPU scheduler using queue

Longest Job First(Preemptive):

It is similar to an Shortest Job First scheduling(SJF) algorithm.

In this scheduling algorithm, priority is given to the process having the largest burst time remaining.

Longest Job First(Non-Preemptive):

- It is similar to an SJF scheduling algorithm. But, in this scheduling algorithm, priority is given to the process having the longest burst time.
- This is non-preemptive in nature i.e., when any process starts executing, can’t be interrupted before complete execution.

Queues – Implementation of CPU scheduler using queue

Round Robin Scheduling:

- To implement Round Robin scheduling, The processes are kept in the queue of processes.
- New processes are added to the rear of the simple queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1-time quantum, and dispatches the process.
- The process may have a CPU burst of less than 1-time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.
- Otherwise, if the CPU burst of the currently running process is longer than 1-time quantum, the timer will go off and will cause an interrupt to the operating system.
- A context switch will be executed, and the process is put at the rear of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Queues – Implementation of CPU scheduler using queue

Premptive Priority Based Scheduling:

- In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time.

- The One with the highest priority among all the available processes will be given the CPU next.

Priority Based(Non-Preemptive) Scheduling:

- In the Non Preemptive Priority scheduling, The Processes are scheduled according to the priority number assigned to them.

- Once the process gets scheduled, it will run till the completion

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402

Data Structures and its Applications

Dinesh Singh
Department of Computer Science & Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Queues – Implementation of Josephus Problem

Dinesh Singh
Department of Computer Science & Engineering

Queues – Implementation of Josephus Problem

- Josephus Problem : Postulates a group of soldiers surrounded by an overwhelming enemy force. There is no hope of victory without reinforcements. There is one horse available for escape

- The soldiers agree to a pact to determine which of them is to escape and seek help. The soldiers form a circle and a number n is picked from a hat. One of the names is also picked from the hat.

- Beginning with the soldier whose name is picked , they begin to count clockwise around the circle. when the count reaches n, that soldier is removed from the circle and the count begins with the next soldier.

- The process continues so that each time the count reaches n, another soldier is removed from the circle. Any soldier removed from the circle is no longer counted. The last soldier remaining is to take the horse and escape.

Queues – Implementation of Josephus Problem

- The input to the program is the number n and list of names, which is the clockwise ordering of the circle, beginning with the soldier from whom the count is to start.
- The program should print the names in the order that they are eliminated and the name of soldier who esapes.
- For example if n=3 and that there are five solders named A,B,C,D and E. We count three soldiers starting at A, so that C is eliminated first.
- We then begin at D and count D E and back to A. A is eliminated. Then we count B D and E, E is eliminated. And finally B D and B is eliminated.
- D is the one who escapes

Queues – Implementation of Josephus Problem

- Data structure used is a circular list where each node represents one soldier
- To represent the removal of a soldier form the circle, a node is deleted from the circular list.
- Finally one node remains on the list and the result is determined

Queues – Implementation of Josephus Problem

Pseudo code of implementation using circular list

```
read(n)
read(name)
while(all the names are read)
{
        insert name on the circular list
        read(name)
}
while(there is more than one node on the list)
{
        count through n-1 nodes on the list
        print name in the nth node
        delete the nth node
}
print the name of the only node on the list
```

Queues – Implementation of Josephus Problem

Code of implementation using circular list

```
int survivor(struct node **head, int n)
{
 // head is pointer to first node

   struct node *p, *q;
   int i;
   q = p = *head;
   while (p->next != p)
      {
         for (i = 0; i < n - 1; i++)
            {
               q = p;
               p = p->next;
            }
         q->next = p->next;
          printf("%d has been killed.\n", p->num);
          free(p);
          p = q->next;
      }
   *head = p;
   return (p->num);
}
```

Queues – Implementation of Josephus Problem

Pseudo code of implementation using circular queue

```
Enter n
while(all the names are read)
{
  insert name into the queue
  read(name)
}
while( q has one element)
{
   dequeue n-1 names from the queue and enqueue it.
   dequeue the nth name
   print the nth name
 }
 dequeue the only name of the queue
 print the name
```

Data Structures and its Applications

Queues – Implementation of Josephus Problem

Assignment :

Implement the Josephus by using circular queue

Implement Josephus Problem by using linked list

THANK YOU

Dinesh Singh
Department of Computer Science & Engineering

dineshs@pes.edu

+91 8088654402