Applications of linked list (continued): Memory management, memory allocation and de-allocation. First-fit, best-fit and worst-fit allocation schemes Implementation of Stacks and Queues using arrays and linked list, DEQUEUE (double ended queue). Multiple Stacks and Queues, Applications.

Application of Linked List –Memory Management

Memory Management with Bitmaps:

When memory is assigned dynamically, the operating system must manage it. With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied.

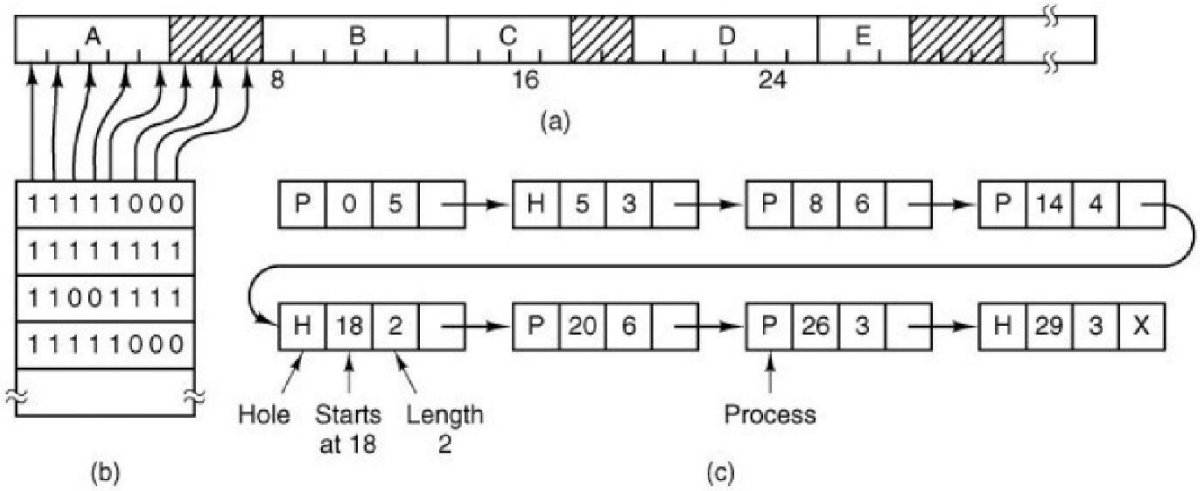
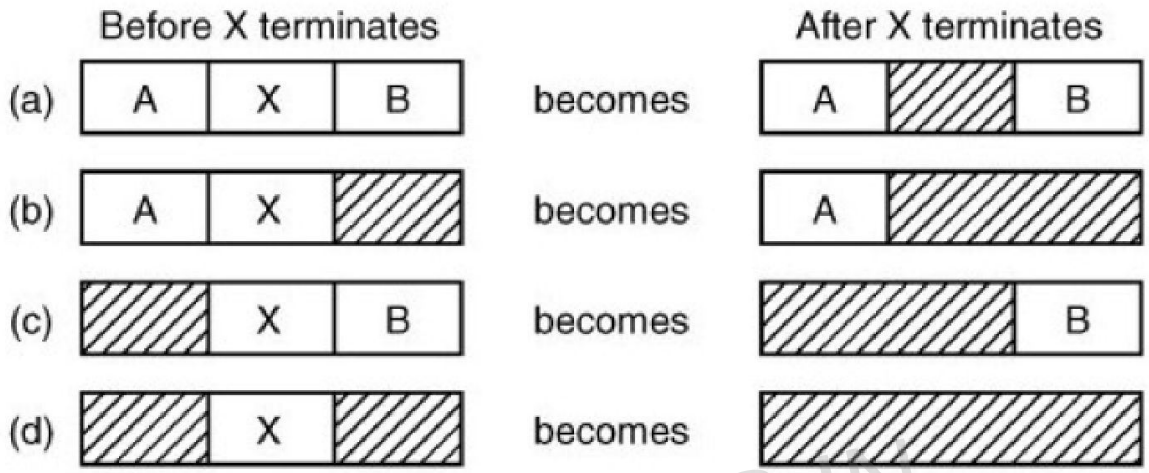


Fig:(a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap. A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem with it is that when

it has been decided to bring a k unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation. Memory Management with Linked Lists Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes.



Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry. In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or very bottom of memory). These may be either processes or holes, leading to the four combinations shown in fig above. When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate.

First Fit: The simplest algorithm is first fit. The process manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically

unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

Next Fit: It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

Best Fit: Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Worst Fit: Always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

STACK

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

The two basic operations associated with stacks are:

- *Push*: is the term used to insert an element into a stack.
- *Pop*: is the term used to delete an element from a stack.

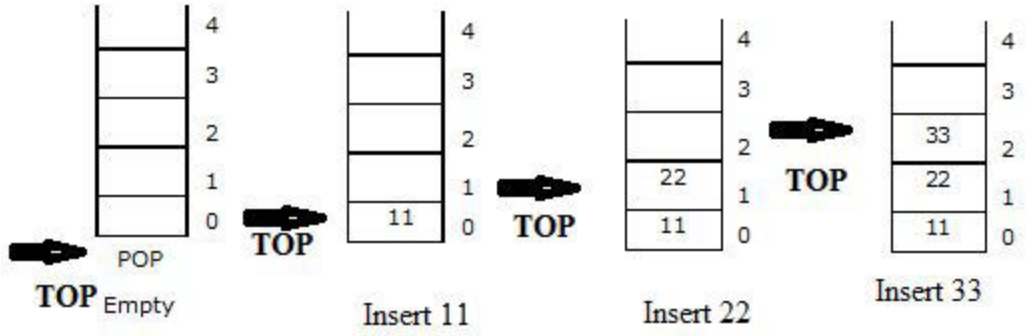
"Push" is the term used to insert an element into a stack. "Pop" is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack:

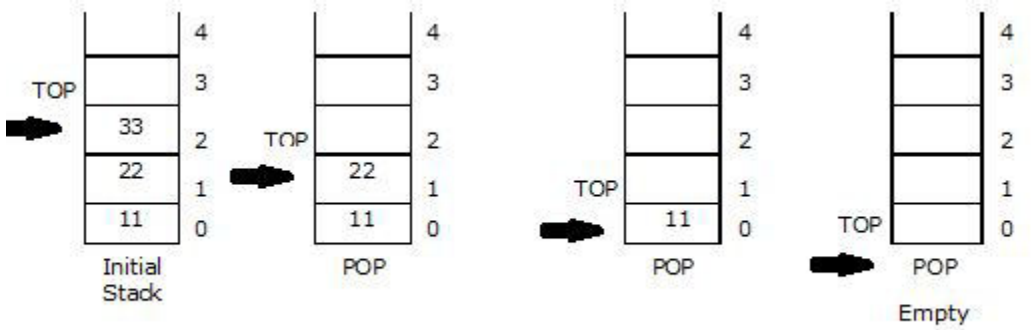
Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the

When an element is added to a stack, the operation is performed by push(). Figure shows the creation of a stack and addition of elements using push().



Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure shows a stack initially with three elements and shows the deletion of elements using pop().



Pop operations on stack

Algorithm for push and pop

2. Set top=-1
3. Read choice ch
4. If ch=1(PUSH), then goto 4.a else 4.b
   a. If top<4 then
      Read the item, set the top=top+1
      Set stack[top]=item
   b. Else Print stack overflow
5. If ch=2(POP), then goto 5.a else 5.b
   a. if top≥0 then
      Set item=stack[top]
      top=top-1
      Print number deleted is item

b. Else Print stack overflow

6. If ch=3, then

   Display the elements in the stack

7. If ch=4, then

   Exit

8. Stop

PROGRAM

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int stack[10],ch,i,item,top=-1;
  clrscr();
  while(1)
  {
    printf("\nMENU:\n1.Push\n2.Pop\n3.Traverse\n4.Exit\nEnter your choice: ");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1:
          if(top<=9)
          {
                printf("\nStack overflow");
          }
          else
          {
```

```
    printf("\nEnter the item: ");
    scanf("%d",&item);
    top=top+1;
    stack[top]=item;
  }

  break;
case 2:
  if(top==-1)
        printf("\nStack underflow");

  else
      {
    item=stack[top];

    printf("\nDeleted item is %d\n",item);
  }

  break;
case 3:
  if(top>=0)
    printf("\nNo elements in stack");

  else
  {
    printf("\nElements are: ");
    for(i=top;i>=0;i--)
```

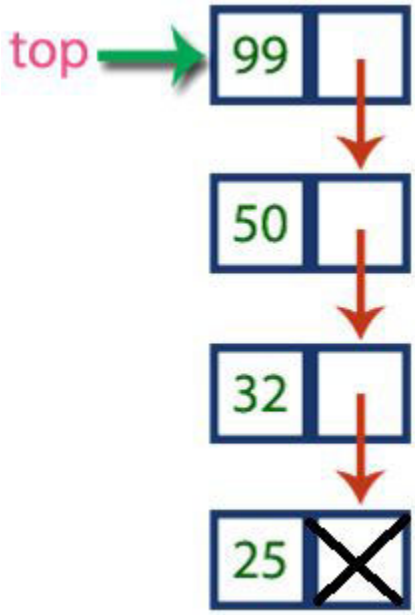
```
            printf("  %d",stack[i]);
          }

          break;
        case 4:
          exit(0);
    }
    getch();
  }
}
```

Stack Using Linked LIST

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means, stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its next node in the list. The **next** field of the first element must be always **NULL**.



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- Step 1: Create a newNode with given value.
- Step 2: Check whether stack is Empty (top == NULL)
- Step 3: If it is Empty, then set newNode → next = NULL.
- Step 4: If it is Not Empty, then set newNode → next = top.
- Step 5: Finally, set top = newNode.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- Step 1: Check whether stack is Empty (top == NULL).
- Step 2: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and terminate the function
- Step 3: If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
- Step 4: Then set 'top = top → next'.
- Step 7: Finally, delete 'temp' (free(temp)).

Program

```
#include<stdio.h>
#include<conio.h>
struct node
{
  int data;


void main()
{
  typedef struct node NODE;
  NODE *top=NULL,*temp,*t;
  int ch,item;
  clrscr();
  while(1)
  {
    printf("\nMENU:\n1.Push\n2.Pop\n3.Display\n4.Exit\nEnter your choice: ");
    scanf("%d",&ch);
    switch(ch)
    {
```

```
case 1:
  temp=(NODE*)malloc(sizeof(NODE));
  printf("\nEnter the item: ");
  scanf("%d",&item);
  temp->data=item;
  if(top==NULL)
  {
    temp->ptr=NULL;
    top=temp;
  }
  else
  {
    temp->ptr=top;
    top=temp;

  break;
case 2:
  if(top==NULL)
    printf("\nDeletion is not possible");
  else if(top->ptr==NULL)
  {
    temp=top;
    top=NULL;
    printf("\nPoped item is %d",temp->data);
    free(temp);
  }
  else
```

```
        {
          temp=top;
         top=top->ptr;
          printf("\nPoped item is %d",temp->data);
          free(temp);
        }
       break;
     case 3:
       if(start==NULL)
         printf("\nStack is empty");
       else
       {
         printf("\nElements are:");
         for(t=top;t!=NULL;t=t->ptr)


       }
       break;
     case 4:
       exit(0);
     default:
       printf("\nWrong Choice");
       break;
    }
    getch();
  }
}
```

Application of Stack

1.Stack Frames

Programs compiled from C make use of a stack frame for the working memory of each procedure or function invocation. When any procedure or function is called, a number of words – the stack frame –is push onto a program stack. When the procedure or function returns, this frame of data is popped off the stack. For example, when a program sends parameters to a function, the parameters are placed ON THE STACK. When the function completes its execution these parameters are popped off from the stack. When a function calls other function the current contents of the caller function are pushed onto the stack with the address of instruction just next to the call instruction, this is done so that after the execution of called function, the compiler can track back the path from where it is sent to the called function.

2. Reversing a String

be

exploited to reverse strings of line of characters. This can be simply thought of simply as pushing the individual characters, and when the complete line is pushed onto the stack, then individual characters of the line are popped off. Because the last inserted character pushed on stack would be the first character to be popped off, the line obviously be removed.

3. Converting INFIX to POSTFIX

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to right.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized.
3. The sub-expression which has been converted into postfix is to be treated as single operand
4. Once the expression is converted to postfix from remove the parenthesis.

Convert the A*B+C/D to postfix form

(A*B)+(C/D)

AB* + (C/D)

AB* + CD/

AB*CD/+

Infix to postfix Conversion

Suppose Q is an arithmetic expression written in notation. This algorithm finds the equivalence postfix expression P

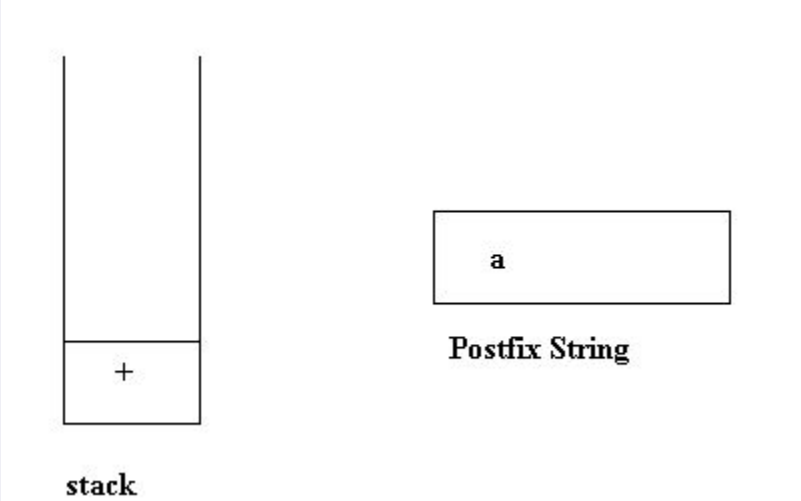
1. Scan Q from left to right and repeat step 2 to 5 for each element of Q until the STACK is empty
2. If an operand is encountered, add it to P.
3. If a left parenthesis is encountered, push it onto STACK.

   a) Repeatedly pop from the STACK and add to P each operator (on the top of the STACK)which has the same precedence or higher precedence than ☼
   b) Add ☼ to STACK
5. If a right parenthesis is encountered, then
   a) Repeatedly pop from the STACK and add to P each operator until a left parenthesis is encountered
   b) Remove the left parenthesis[Do not add left parenthesis to P]
6. Exit

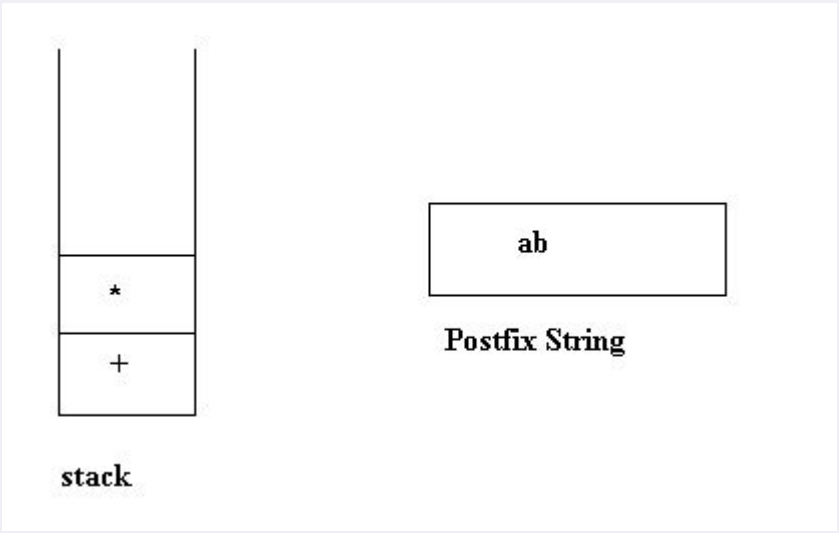
Let us see how the above algorithm will be implemented using an example.

Infix String : a+b*c-d

Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.

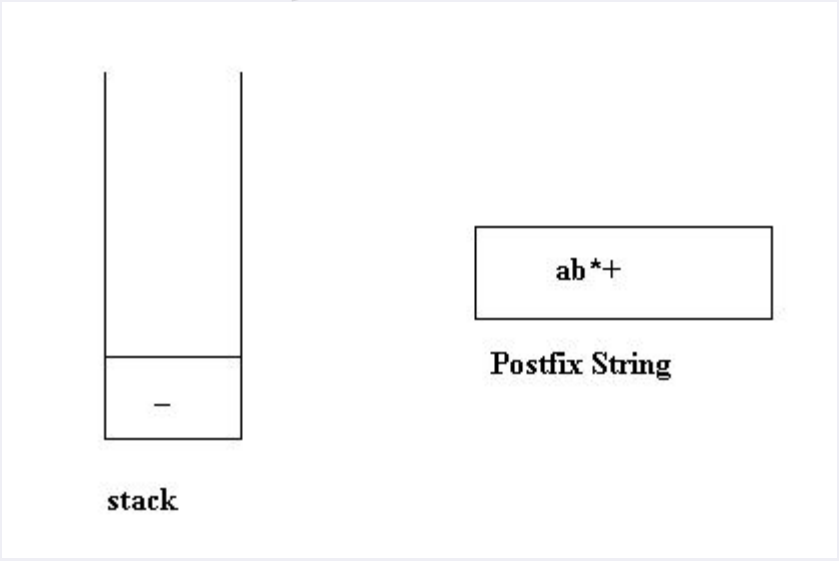


Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.

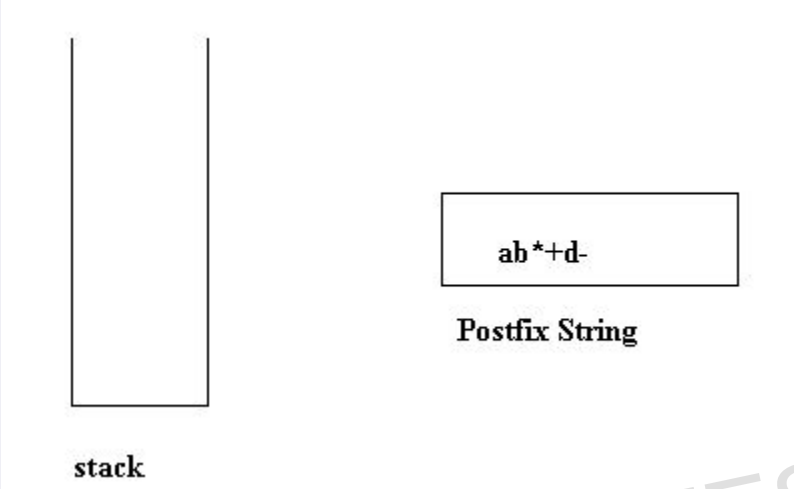


The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the

add it to the Postfix string. The '-' will be pushed to the stack.



Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :



End result :

- Infix String : a+b*c-d
- Postfix String : abc*+d-

Postfix evaluation

The reason to convert infix to postfix expression is that we can compute the answer of postfix expression easier by using a stack.

This algorithm finds the VALUE of the arithmetic expression P written in postfix notation. The following

algorithm, which uses a STACK to hold operands, evaluates P.

1. Scan P from left to right and repeat step 2 and 3 for each element of P until the end of the string.
2. If an operand is encountered, put it on stack.
3. If an operator ☼ is encountered, then
    a) Remove the two top elements of STACK , where A is the top element and B is the next top element
    b) Evaluate B☼A
    c) Place the result of (b) back on \STACK
    [End of IF structure]
4. Set VALUE equal to the top element on stack

Example :

Let us see how the above algorithm will be imlemented using an example.

Postfix String : 123*+4-

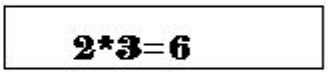
Initially the Stack is empty. Now, the first three characters scanned are 1,2 and 3, which are operands. Thus they will be pushed into the stack in that order.

stack

expression

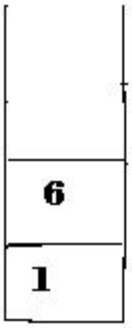
Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.

stack



expression

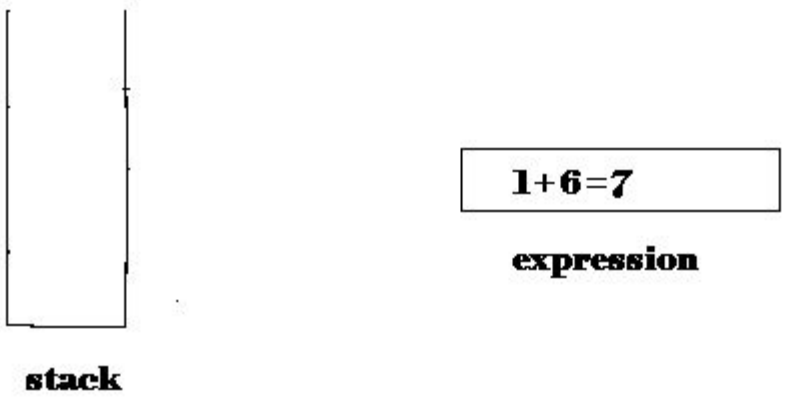
The value of the expression(2*3) that has been evaluated(6) is pushed into the stack.



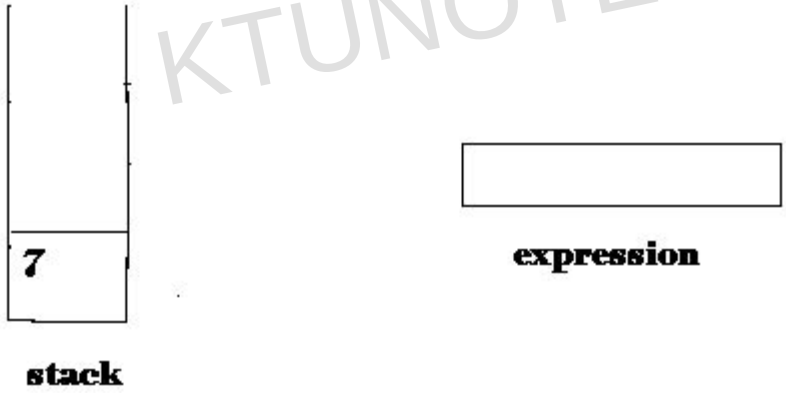
stack

expression

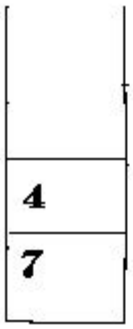
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.



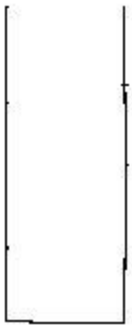
Next character scanned is "4", which is added to the stack.



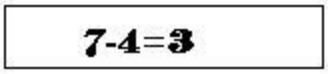
stack

expression

Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.

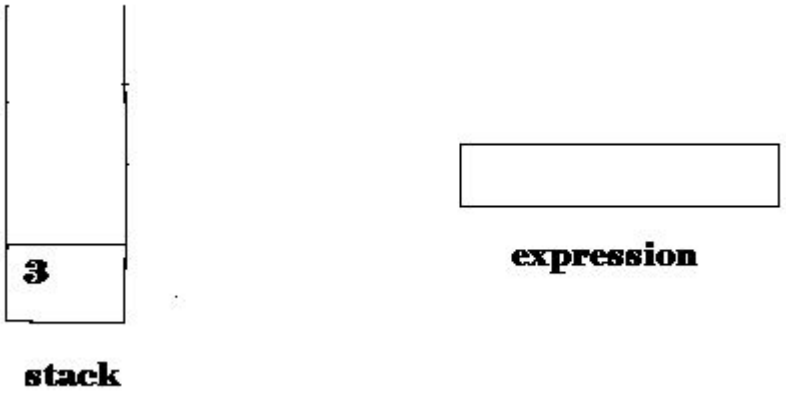


stack



expression

The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result :

- Postfix String : 123*+4-
- Result : 3

Queue

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek( )** function is often used to return the value of first element without deleting an element.

Queue using array

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then increment 'front' value by one and then display the value at 'front' position as deleted element.

Queue Operations

Enqueue-Insert value into the queue

1. Start
2. If queue is full (rear>=size-1) then queue is full otherwise goto step 3
    a) set front=rear=0
    b) insert element into the rear position
4. Increment rear and insert the element into the rear
5. Stop

Dequeue- delete an element from queue

1. Start
2. If queue is empty ((front=-1 and rear==-1)or (front=rear+1)) then print queue is empty otherwise goto step 3
3. Increment the front value by one (front++). Then display queue[front]as deleted element
4. Stop

Program

```
#include<stdio.h>
```

```
#include<conio.h>
#include<stdlib.h>
int main()
{
   int ch, front=-1, rear=-1, q[10], item, i;
   while(1)
   {
      printf("\n\t QUEUE\n\n 1. Insert \n 2. Delete \n 3. Display \n 4. Exit\n Enter ur choice: ");
      scanf("%d", &ch);
      switch(ch)
      {
         case 1:
               if(rear==9)
               {
                  printf("Queue is full ! ");
               }
               else
               {
                  if(front==-1 && rear==-1)
                  {
```

```
            front=0; rear=0;
        }
        else
            rear++;
        printf("Enter the inserted item :  ");
        scanf("%d", &item);
        q[rear]=item;
    }
    break;
case 2:
    if((front==-1 && rear==-1) || front==rear+1)
    {
        printf("Queue is empty!");
    }
        else
        {
            item=q[front];
            front++;
            printf("Deleted item is : %d", item);
        }
```

```
            }
            break;
        case 3:
            if((front==-1 && rear==-1) || front==rear+1)
            {
               printf("Queue is empty ! ");
            }
            else
            {
               printf("\nElements are :");
               for(i=front; i<=rear; i++)
                     printf("%d  \t", q[i]);
            }
            break;
        case 4:
            exit(0);
            break;
    }
}
getch();
```

}

Queue Using Linked List

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

To implement queue using linked list, we need to set the following things before implementing actual operations.

- Step 1: Include all the header files which are used in the program. And declare all the user defined functions.
- Step 2: Define a 'Node' structure with two members data and next.
- Step 3: Define two Node pointers 'front' and 'rear' and set both to NULL.
- Step 4: Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- Step 1: Create a newNode with given value and set 'newNode → next' to NULL.
- Step 2: Check whether queue is Empty (rear == NULL)
- Step 3: If it is Empty then, set front = newNode and rear = newNode.
- Step 4: If it is Not Empty then, set rear → next = newNode and rear = newNode.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- Step 1: Check whether queue is Empty (front == NULL).
- Step 2: If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function

nt'.

- Step 4: Then set 'front = front → next' and delete 'temp' (free(temp)).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- Step 1: Check whether queue is Empty (front == NULL).
- Step 2: If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
- Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
- Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next !=NULL).

- Step 4: Finally!_Display 'temp → data ---> NULL'.

Program

```
#include<stdio.h>
struct node
{
   int data;
   struct node *ptr;
};
int main()
{
   typedef struct node NODE;
   NODE *front=NULL,*rear=NULL, *temp, *p;
   int i, ch, pos, item;
   while(1)
   {
         printf("\n\n\tQueue Using Linked List\n1.Insert\n2.Delete\n3.Display\n4.  Exit\n\nEnter your
            choice:  ");
      scanf("%d", &ch);
      switch(ch)
```

```
{
  case 1:
        temp=(NODE*)malloc((sizeof(NODE)));
        printf("Enter the data to be inserted:  ");
        scanf("%d", &temp->data);

        if(rear==NULL)
        {
           temp->ptr=NULL;
           front=rear=temp;
        }
        else
        {
           p=front;
           while(p!=rear)
           {
              p=p->ptr;
           }
           p->ptr=temp;
           temp->ptr=NULL;
```

```
        rear=temp;
      }
      break;
case 2:
      if(front==NULL)
      {
        printf("No elements to delete!");
      }
      else
      {
        if(front->ptr==NULL)
        {
          temp=front;
          item=temp->data;
          printf("Deleted element is %d", temp->data);
          free(temp);
          front=rear=NULL;
        }
        else
        {
```

```
            temp=front;
            item=temp->data;
            front=front->ptr;
            free(temp);
            printf("Deleted element: %d", item);
        }
    }
    break;
case 3:
    if(front==NULL)
    {
        printf("No elements");
    }
    else
    {
        printf("\nElements are: ")
        p=front;
        while(p!=rear)
        {
            printf("%d  ", p->data);
```

```
                p=p->ptr;
              }
              printf("%d", p->data);
            }
            break;
        case 4:
            exit(0);
      }
   }
   getch();
}
```

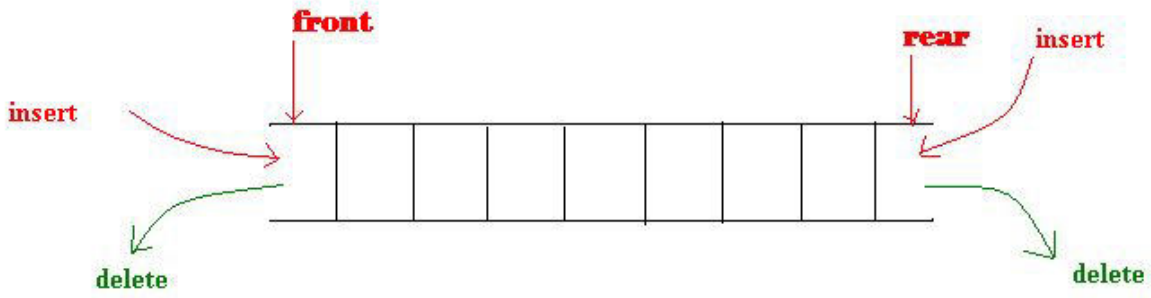
Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Double ended Queue (Deque)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

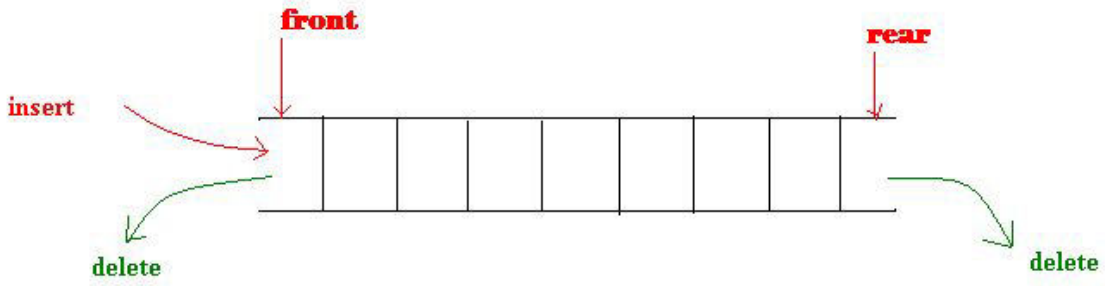


Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

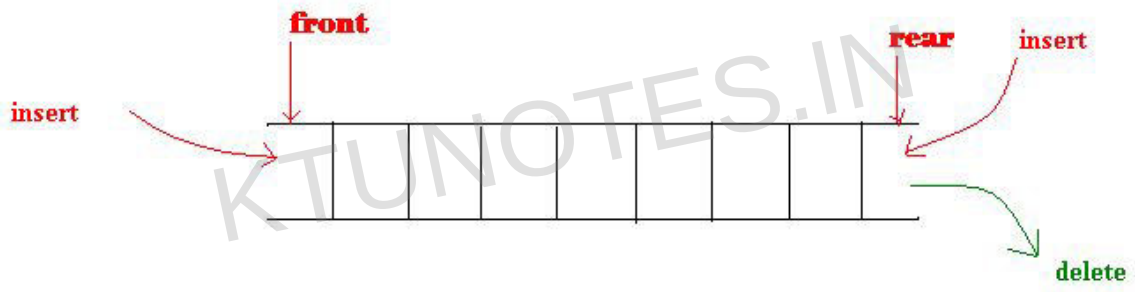
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Algorithm for Insertion at rear end

Step -1: [Check for overflow]

if(rear==MAX)

Print("Queue is Overflow");

return;

Step-2: [Insert element]

else

rear=rear+1;

q[rear]=no;

[Set rear and front pointer]

if rear=-1

rear=front=0;

Step-1 : [Check for the front position]

if(front==0)

-

Print ("Cannot add item at front end");

return;

Step-2 : [Insert at front]

else

front=front-1;

q[front]=no;

Step-3 : Return

Algorithm for Deletion from front end

Step-1 [Check for front pointer]

if front=-1

print(" Queue is Underflow");

return;

Step-2 [Perform deletion]

print("Deleted element is",no);

[Set front and rear pointer]

if front=rear

front=rear=-1;

else

front=front+1;

Step-3 : Return

Algorithm for Deletion from rear end

Step-1 : [Check for the rear pointer]

if rear=0

print("Cannot delete value at rear end");

return;

Step-2: [perform deletion]

else

if front= rear

front=rear=-1;

else

rear=rear-1;

print("Deleted element is",no);

Step-3 : Return

Multiple Stacks

This C Program Implements two Stacks using a Single Array & Check for Overflow & Underflow. A Stack is a linear data structure in which a data item is inserted and deleted at one record. A stack is called a Last In First Out (LIFO) structure. Because the data item inserted last is the data item deleted first from the stack.

To implement two stacks in one array, there can be two methods.

First is to divide the array in to two equal parts and then give one half two each stack. But this method wastes space.

So a better way is to let the two stacks to push elements by comparing tops of each other, and not up to one half of the array.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int n,top1,top2,ch=1,a,i,arr[100];
printf("Enter size of array you want to use\n");
scanf("%d",&n);
top1=-1;
top2=n/2;
while(ch!=0)
{
printf("What do u want to do?\n");
printf("1.Push element in stack 1\n");
printf("2.Push element in stack 2\n");
printf("3.Pop element from stack 1\n");
printf("4.Pop element from stack 2\n");
```

```
printf("5.Display stack 1\n");
printf("6.Display stack 2\n");
printf("0.EXIT\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
{
printf("Enter the element\n");
scanf("%d",&a);
if(top1==n/2-1)
printf("Stack1 is full");
else
{
top1++;
arr[top1]=a;
}
break;
}
case 2:
{
printf("Enter the element\n");
scanf("%d",&a);
if(top2==n)
printf("Stack 2 is full\n");
else
{
```

```
top2++;
arr[top2]=a;
}
break;
}
case 3:
{
if(top1==-1)
printf("Stack1 is empty\n");
else
{
a=arr[top1];
top1--;
printf("%d\n",a);
}
break;
}
case 4:
{
if(top2==n/2)
printf("Stack2 is empty\n");
else
{
a=arr[top2];
top2--;
printf("%d\n",a);
}
```

```
break;
}
case 5:
{
if(top1==-1)
printf("Stack1 is empty\n");
else
{
printf("Stack1 is-->>\n");
for(i=0;i<=top1;i++)
printf("%d ",arr[i]);
printf("\n");
}
break;
}
case 6:
{
if(top2==n/2)
printf("Stack2 is empty\n");
else
{
printf("Stack2 is-->>\n");
for(i=(n/2+1);i<=top2;i++)
printf("%d ",arr[i]);
printf("\n");
}
break;
```

```
}
case 0:
break;
}
}
    }
```

Multiple Queues

We can maintain two queues in the same array which is possible. If one grows from position 0 of the array and the other grows from the last position, queue refers to the data structure where the elements are stored such that a new value is inserted at the rear end of the queue and deleted at the front end of the queue.

In order to preserve two queues, there should be two front and two rear of the two queues. Both the queues can grow upto any extent from 0^{th} position to maximum. Hence there should

front=-1 rear=-1

front2=n/2-1 rear2=n/2-1

Insert into First Queue

```
if(rear1=n/2-1)
{
printf("First Queue is full\n");
}
else
```

```
{
printf(“Enter the element to be inserted”);
scanf(“%d”,&item);
If((front1==-1)&&(rear1==-1))
{
front1=rear1=0;
}
else
{
rear1++;
}
Q[rear1]=item;
}
```


Insert into Second Queue

```
if(rear2=n/-1)
{
printf(“Second Queue is full\n”);
}
else
```

```
{
printf("Enter the element to be inserted");
scanf("%d",&item);
If((front2==n/2-1)&&(rear1==n/2-1))
{
front2=rear2=n/2;
}
else
{
rear2++;
}
Q[rear2]=item;
}
```


Delete from First Queue

```
if(front1==-1)
{
printf("First Queue is empty\n");
}
else
```

```
{
item=Q[front1];
printf("The deleted element in the first queue is %d",item);
If((front1==rear1)
{
front1=rear1=-1;
}
else
{
front1++;
}
}
```


Delete from Second Queue

```
if(front2==n/2-1)
{
printf("Second Queue is empty\n");
}
else
{
```

```
item=Q[front2];

printf("The deleted element in the first queue is %d",item);

If((front2==rear2)

{

front2=rear2=n/2-1;

}

else

{

front2++;

}

}
```