PUNJAB UNIVERSITY REGIONAL CENTER

DATA STRUCTURES

Programs File

Prepared by:

1803, MCA

Submit to:

MR. MOHINDER KUMAR

TABLE OF CONTENTS

I. ARRAYS

II. LINKED LISTS

III. STACKS

III. QUEUES

Arrays - Array Structures Introduction

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

Traversing - Simple array traversing

We will do a very basic traversing which includes mere executing loop over every element in the array and print that element.

Algorithm

```
display()
 1. i ← 0
 2. Repeat while i < n
     A. Write “ array[i] ”
     B.  i ← i +1
     End of loop
```

Program

```
void display() {
    int i;
    for( i = 0; i < n; printf("%d ",array[i]), i++);
}
```

Insert element - Array Structures

Below program is capable of inserting new elements in an array. It basically displaces other elements to make position for new element to be inserted. n = Number of elements present.

Algorithm

```
insert(x, loc)
 1. i ← n-1
 2. Repeat while i >= loc
    A. a[i+1] ← a[i]
    B. i ← i-1
    end loop

 3. a[loc] ← x

 4. n ← n+1
```

Program

```
void insert( int x, int loc) {

    int i;

    for( i = n-1; i >= loc; a[i+1] = a[i],i--);

    a[loc]=x;

    n++;

}
```

Delete element - Array Structures

Below program is capable of deleting a record which is located at location holded by loc variable.

```
delete(loc)

 1. i ← loc

 2. Repeat while i < n-1
   A. a[i] ← a[i+1]
   B. i ← i+1
  end loop

 4. n ← n-1
```

```
void insert( int x, int loc) {

    int i;

    for( i = n-1; i >= loc; a[i+1] = a[i],i--);

    a[loc]=x;

    n++;

}
```

Searching – Searching arrays introduction

Well, to search an element in a given array, there are two popular algorithms available:

- Linear Search
- Binary Search

Linear Searching – Searching array 01/02

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1.

Algorithm

```
linearSearch(a, n, x)

 1. i ← 0

 2. Repeat while i < n

    A. If a[i] = x then

         return i

        end if

    B. i ← i+1

 3. return -1
```

Program

```
int linearSearch(int a[], int n, int x) {

    int i;

    for (i = 0; i < n; i++)

        if (a[i] == x) return i;

    return -1;

}
```

Binary Searching - Array Structures

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array.

If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Algorithm

```
binarySearch(a, l, r, x)

 1. mid ← 0

 2. if r >= l then

     A. mid ← l + (r-1)/2

     B. if a[mid] = x then
          return mid
        end if

     C. if a[mid] > x then
          return binarySearch(a, l, mid-1, x)
        end if

     D. binarySearch(a, mid+1, r, x)

   end if
```

Program

```
int binarySearch(int a[], int l, int r, int x) {

    int mid;

    if (r >= l) {

        mid = l + (r - l) / 2;

        if (a[mid] == x) return mid;

        if (a[mid] > x) {

          return binarySearch(a, l, mid - 1, x);

        }

        return binarySearch(a, mid + 1, r, x);
    }

    return -1;

}
```

Bubble Sort – Sorting algorithms 01/04

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

Algorithm

```
bubbleSort(a, n)

 1. i ← 0
 2. j ← 0

 3. Repeat while i < n-1

    A. Repeat while j < n-i-1

        I. If a[j] > a[j+1] then

           swap(a[j], a[j+1])

          end if

        II.  j ← j+1

       end of loop

    B.  i ← i +1

   end of loop
```

Program

```c
void bubbleSort(int a[], int n) {

    int i,j;

    for(i=0;i<n-1;i++) {

        for(j=0;j<n-i-1;j++) {

            if(a[j] > a[j+1]) swap(&a[j], &a[j+1]);

        }

    }

}
```

Merge Sort – Sorting algorithms 03/04

Algorithm

```
merge(a, b, m, e)

        1. lb ← b, rb ← m+1, tb ← b

        2. while lb <= m and rb <= e do
                if a[lb] < a[rb] then
                        temp[tb] ← a[lb];
                        tb ← tb+1
                        lb ← lb+1
                else
                        temp[tb]=a[rb];
                        tb ← tb+1
                        rb ← rb+1;
                end if
         end if

        3. while lb <= m do
                temp[tb]<-a[lb];
                tb ← tb+1
                lb ← lb+1
         end while

        4. while rb <= e do
                temp[tb] = a[rb];
                tb ← tb+1
                rb ← rb+1
         end while

        5. while b <= e do
                a[b] ← temp[b];
                b ← b+1
         end while

mergesort(a, l, h)

        1. mid <-0

        2. if l < h then
               mid ← (l + h) / 2
               mergesort(a,1,mid);
               mergesort(a,mid+1,h);
               merge(a,0,mid,h);
         end if
```

Program

```
void merge(int a[],int b,int m,int e) {
        int lb = b, rb = m+1, tb = b;

        while((lb<=m)&&(rb<=e)) {

                if(a[lb]<a[rb]) {
                        temp[tb]=a[lb];
                        tb++;
                        lb++;
                } else {
                        temp[tb]=a[rb];
                        tb++;
                        rb++;
                }

        }

        while(lb<=m) {
                temp[tb]=a[lb];
                tb++;
                lb++;
        }

        while(rb<=e) {
                temp[tb] = a[rb];
                tb++;
                rb++;
        }

        while(b<=e) {
                a[b] = temp[b];
                b++;
        }

}

void mergesort(int a[],int l,int h) {
        int mid;

        if(l < h) {

                mid = (l + h) / 2;
                mergesort(a,1,mid);
                mergesort(a,mid+1,h);
                merge(a,0,mid,h);

        }

}
```

Heap Sort – Sorting algorithms 04/04

```
insert(i)
  1. if n>=m then
       Write " Overflow "
   else
              n ← n+1
              a[n] ← i
              t ←  n
              pt ← n / 2
              while pt >= 1 and a[t] > a[pt] then
                   temp ← a[t]
                   a[t] ← a[pt]
                   a[pt] ← temp
                   t ← pt
                   pt ← pt/ 2
              end while
         end if

shiftdown(lo)
         1. t ← 1
         2. flag ← 0
         3. while flag = 0 then
              pt ← t
              if 2*pt <= end and a[2*pt] > a[t] then
                   t ← 2*pt end if
              if (2*pt) + 1 <= end and a[2*pt+1] > a[t] then
                   t ← 2*pt+1 end if
              if t = pt then
        flag ← 1
      else
                   temp ← a[t]
                   a[t] ← a[pt]
                   a[pt] ← temp
              end if
         end while
delete(loc)
         1. item ← a[loc]
         2. a[loc] ← a[n]
         3. end ← n
         4. n ← n-1
         5. shiftdown(loc)
traverse()
         1. for i=1 until i<=n do
              Write a[i]
              i ← i+1
         end for
heapSort()
         1. end ← n
         2. while end > 0 do
              temp ← a[1]
              A[1] ← a[end]
              a[end] ← temp
              end ← end-1
              shiftdown(1)
   end while
```

Program

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

static int a[8],m=8,n=0,t,pt,temp,item,flag,end;

void insert(int i) {
    if(n>=m) {
     printf("\nOverflow");
  } else {
       n++;
       a[n] = i;
       t = n;
       pt = n / 2;
       while(pt >= 1 && a[t] > a[pt]) {
           temp = a[t];
           a[t] = a[pt];
           a[pt] = temp;
           t = pt;
           pt = pt/ 2;
       }
    }
}
void shiftdown(int lo) {
    t = 1;
    flag = 0;
    while(flag==0) {
       pt = t;
       if((2*pt) <= end && a[2*pt]>a[t])
           t=2*pt;
       if((2*pt)+1 <= end && a[2*pt+1]>a[t])
           t=2*pt+1;
       if(t==pt) {
      flag = 1;
    } else {
           temp = a[t];
           a[t] = a[pt];
           a[pt] = temp;
       }
    }
}
void delete(int loc){
    item = a[loc];
    a[loc] = a[n];
    end = n;
    n--;
    shiftdown(loc);
}
void traverse() {
    int i;
    for(i=1;i<=n;i++) printf("\t%d", a[i]);
}
void heapSort() {
    end = n;
    while (end>0){
       temp = a[1];
       a[1] = a[end];
       a[end] = temp;
       end = end-1;
       shiftdown(1);
  }
}
```

Linked lists – Data Structures 02/05

Linked List is a very commonly used linear data structure which consists of group of nodes in a sequence.

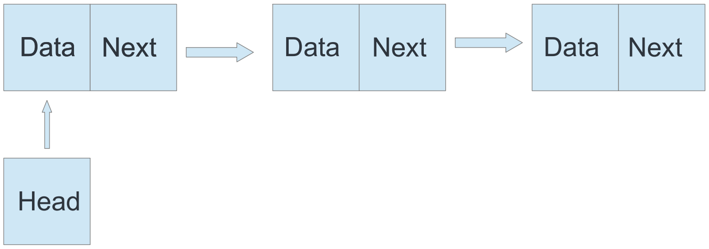
Each node holds its own data and the address of the next node hence forming a chain like structure.

Linked Lists are used to create trees and graphs.

Single linked lists – Linked lists 01/03

Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are insertion, deletion and traversal.



Traversing – Basic program

Algorithm

```
displayList()

  1. temp ← head;

  2. While temp != NULL Do

     A. if temp = head then Write " data[temp]  "

        else Write " => data[temp] "

     B. temp ← next[temp];

     end while
```

Program

```
// Structure declaration for reference
struct item {
        int data;
        struct item * next;
};

void displayList() {

    temp = head;

    while(temp) {

        if(temp == head) printf("%d", temp->data);

        else printf(" => %d", temp->data);

        temp = temp->next;

    }

}
```

Insert in the Begining – Insert elements in Single linked list 01/03

Below program is capable of inserting new elements in the beginning of single linked list.

Algorithm

```
insBeg(x)

 1. node ← new

 2. data[node] ← x

 3. next[node] ← head

 4. head ← node
```

Program

```
// Insert in the begining
void insBeg(int x) {

  node = (struct item *) malloc(sizeof(struct item));

  node->data = x;

  node->next = head;

  head = node;

}
```

Insert at End – Insert elements in Single linked list 02/03

Below program is capable of inserting new elements in the end of single linked list.

```
insEnd(x)

  1. node ← new

  2. data[node] ← x

  3. next[node] ← NULL

  4. temp ← head;

  5. while temp != NULL do

     A. if next[temp] != NULL then
        temp ← next[temp]
       else
        next[temp] ← node;
        break while loop
       end if


   end while loop
```

```
// Insert in the end
void insEnd(int x) {

  node = (struct item *) malloc(sizeof(struct item));

  node->data = x;

  node->next = NULL;

  temp = head;

  while(temp) {

      if(temp->next) {
          temp = temp->next;
      } else {
          temp->next = node;
          break;
      }

  }

}
```

Insert anywhere – Insert elements in Single linked list 03/03

Below program is capable of inserting new elements at any place of single linked list.

Algorithm

```
insAt(x, pos)

  1. i ← 0 , end ← 0

  2. end ← getNumberOfElements()

  3. if pos = 1 then

       return insBeg(x)

     else if  pos <= end then

       node ← new
       data[node] ← x

       temp ← head;

       for i=2 until i < pos do

           temp ← next[temp]

       end for

       next[node] ← next[temp]

       next[temp] ← node

    else

       return insEnd(x)

    end if
```

Program

```
// Insert anywhere in the list
void insAt(int x, int pos) {

  int i = 0, end;

  // This returns total number of elements
  end = getNumberOfElements();

  if(pos == 1) {

      return insBeg(x);

  } else if (pos <= end) {

      node = (struct item *) malloc(sizeof(struct
item));

      node->data = x;

      temp = head;

      for(i=2;i<pos;i++) {
          temp = temp->next;
      }

      node->next = temp->next;

      temp->next = node;

  } else {
      return insEnd(x);
  }

}
```

Delete from Begining – Delete elements from Single linked list 01/02

Below program is capable of deleting elements from the beginning of single linked list.

Algorithm

```
delBeg()

  1. head ← next[head]
```

Program

```
// Delete from begining
void delBeg() {
    head = head->next;
}
```

Delete from End – Delete elements from Single linked list 02/02

Below program is capable of deleting elements from the end of single linked list.

Algorithm

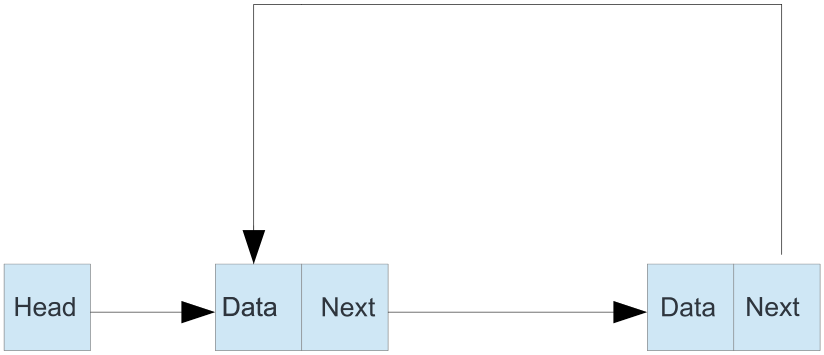
```
delEnd()

  1. temp ← head;

  2. if temp != NULL then

    if next[temp] != NULL then

      while next[next[temp]] != NULL do

        if temp != NULL then
          temp ← next[temp]
        else
          temp ← NULL
        end if

      end while

      next[temp] ← NULL

    else

      head ← NULL

    end if

  end if
```

Program

```
// Delete from the end
void delEnd() {

    temp = head;

    if(temp) {

        if(temp->next) {

            while(temp->next->next) {
                if(temp->next) {
                    temp = temp->next;
                } else {
                    temp = NULL;
                }
            }

            temp->next = NULL;

        } else {

            head = NULL;

        }

    }

}
```

Circular Linked lists – Linked lists 02/03

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



Search – Stores location in loc variable

Algorithm

```
search( x)

  1. temp ← header.next;

  2. while temp != head do

            if data[temp] = x then
              loc ← temp
              break while loop
            end if

            prv ← temp

      temp ← next[temp]

    end while loop
```

Program

```
// Structure declaration for reference
struct item {
        int data;
        struct item * next;
        struct item * prev;
} *node,*head,*temp,*loc,*prv,header;

void search(int x) {

    temp = header.next;

    while(temp!=head) {

           if(temp->data==x) {
               loc = temp;
               break;
           }

           prv = temp;

           temp = temp->next;

    }

}
```

Insert in the Begining – Insert elements in Circular linked list 01/03

Below program is capable of inserting new elements in the beginning of circular linked list.

Algorithm

```
insBegining(x)

  1. node ← new

  2. data[node] ← x

  3. next[node] ← header.next

  4. header.next ← node
```

Program

```c
// Insert in the begining
void insBegining(int x) {

  node = (struct item *) malloc(sizeof(struct item));

  node->data = x;

  node->next = header.next;

  header.next = node;

}
```

Insert at End – Insert elements in Circular linked list 02/03

Below program is capable of inserting new elements in the end of circular linked list.

Algorithm

```
insEnd(x)

  1. node ← new

  2. data[node] ← x

  3. temp ← header.next

  4. while temp != head do
       loc ← temp
       temp ← next[temp]
     end while loop

  5. next[loc] ← node
```

Program

```c
// Insert in the end
void insEnd(int x){

  node = (struct item *) malloc(sizeof(struct item));

  node->data = x;

  temp = header.next;

  while(temp != h) {
      loc = temp;
      temp = temp->next;
  }

  loc->next = node;

}
```

Insert middle – Insert elements in Circular linked list 03/03

Below program is capable of inserting new elements at any place of circular linked list using search algorithm.

Algorithm

```
insMiddle(x, node_data)

  1. search(x)

  2. node ← new

  3. data[node] ← node_data

  4. next[node] ← next[loc]

  5. next[loc] ← node
```

Program

```
// Insert anywhere in the list
void insMiddle(int x, int node_data) {

  // Search and save position in 'loc'
  search(x);

  node = (struct item *) malloc(sizeof(struct item));

  node->data = node_data;

  node->next = loc->next;

  loc->next = node;

}
```

Delete begining – Delete elements from Circular linked list 01/02

Below program is capable of deleting elements from beginning of circular linked list.

```
delBegining()

  1. if header.next != NULL then

       header.next ← next[header.next]

     end if
```

```
// Delete from begining
void delBegining() {

    if(header.next) {
        header.next = header.next->next;
    }

}
```

Delete middle – Delete elements from Circular linked list 02/02

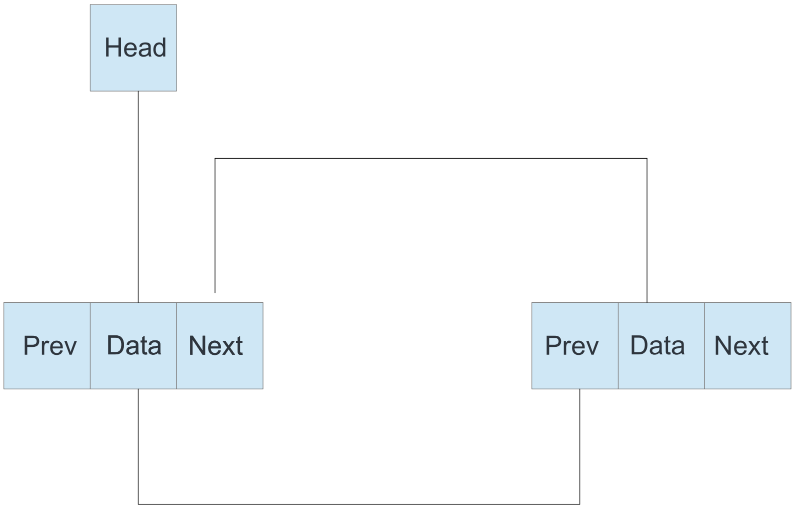
Below program is capable of deleting elements from middle of circular linked list using search algorithm.

```
delMiddle(x)

  1. search(x)

  2. next[prv] ← next[loc]
```

```
// Delete from middle using search()
void delMiddle(int x) {

    search(x);

    prv->next = loc->next;

}
```

Double Linked lists – Linked lists 03/03

In a doubly linked list, each node contains a data part and two addresses, one for the previous node and one for the next node.



Traversing – Forwar based on next

Algorithm

```
displayList()

  1. temp ← head;

  2. While temp != NULL Do

    A. if temp = head then Write " data[temp]  "

       else Write " => data[temp] "

    B. temp ← next[temp];

    end while
```

Program

```
// Structure declaration for reference
struct item {
        int data;
        struct item * next;
        struct item * prev;
};

void displayList() {

    temp = head;

    while(temp) {

        if(temp == head) printf("%d", temp->data);

        else printf(" => %d", temp->data);

        temp = temp->next;

    }

}
```

Insert in the Begining – Insert elements in Double linked list 01/03

Below program is capable of inserting new elements in the beginning of double linked list.

Algorithm

```
insBeg(x)

  1. node <-new

  2. data[node] ← x

  3. next[node] ← head

  4. prev[node] ← NULL

  5. if head != NULL
       prev[head] ← node;
     end if

  6. head ← node;
```

Program

```
void insBeg(int x) {

  node = (struct item *) malloc(sizeof(struct item));

  node->data = x;

  node->next = head;

  node->prev = NULL;

  if(head) {
      head->prev = node;
  }

  head = node;

}
```

Insert at End – Insert elements in Double linked list 02/03

Below program is capable of inserting new elements in the end of double linked list.

Algorithm

```
insEnd(x)

  1. node ← new

  2. data[node] ← x
  3. next[node] ← NULL
  4. prev[node] ← NULL

  5. temp ← head

  6. while temp != NULL do

       if next[temp] != NULL then
         temp ← next[temp]
       else
         next[temp] ← node
         prev[node] ← temp
         break while loop
       end if

    end while loop
```

Program

```
// Insert in the end
void insEnd(int x) {

  node = (struct item *) malloc(sizeof(struct item));

  node->data = x;

  node->next = node->prev =  NULL;

  temp = head;

  while(temp) {

      if(temp->next) {
          temp = temp->next;
      } else {
          temp->next = node;
          node->prev = temp;
          break;
      }
  }

}
```

Insert anywhere – Insert elements in Double linked list 03/03

Below program is capable of inserting new elements at any place of double linked list.

Algorithm

```
insAt(x, pos)

  1. i ← 0 , end ← 0

  2. end ← getNumberOfElements()

  3. if pos = 1 then

       return insBeg(x)

    else if  pos  > end then

       return insEnd(x)

    else

       node ← new
       data[node] ← x
       prev[node] ← NULL
       next[node] ← NULL

       temp ← head;

       for i=2 until i < pos do

          temp ← next[temp]

       end for

       next[node] ← next[temp]

       next[temp] ← node

       prev[node] ← temp

    end if
```

Program

```
// Insert anywhere in the list
void insAt(int x, int pos) {

  int i = 0, end;

  // This returns total number of elements
  end = getNumberOfElements();

  if(pos == 1) {

      return insBeg(x);

  } else if (pos > end) {

  } else {

      node = (struct item *) malloc(sizeof(struct
item));

      node->data = x;

      node->next = node->prev = NULL;

      temp = head;

      for(i=2;i<pos;i++) {
          temp = temp->next;
      }

      node->next = temp->next;

      temp->next = node;

      node->prev = temp;

  }

}
```

Delete from Begining – Delete elements from Double linked list 01/02

Below program is capable of deleting elements from the beginning of Double linked list.

Algorithm

```
delBeg()

  1. if next[head] != NULL then
       prev[next[head]] ← NULL
       head ← next[head]
     else
       head ← NULL
     end if
```

Program

```
// Delete from begining
void delBeg() {

    if(head->next) {
        head->next->prev = NULL;
        head = head->next;
    } else {
        head = NULL;
    }

}
```

Delete from End – Delete elements from Double linked list 02/02

Below program is capable of deleting elements from the end of double linked list.

Algorithm

```
delEnd()

  1. temp ← head;

  2. if temp != NULL then

       if next[temp] != NULL then

         while next[next[temp]] != NULL do
           temp ← next[temp]
         end while loop

         next[temp] ← NULL

       else
         head ← NULL
       end if

     end if
```

Program

```
// Delete from the end
void delEnd() {

    temp = head;

    if(temp) {
        if(temp->next) {
            while(temp->next->next) {
                temp = temp->next;
            }

            temp->next = NULL;
        } else {
            head = NULL;
        }
    }

}
```

Stacks structures – Data structures 03/05

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.

Structure and Operations – Stacks

Stacks can be created using classes, linked lists and arrays. We will use arrays for creating and maintaing our stacks. Let's take a[5] as our initial stack.

Push operations is basically used to push elements in the stack. Newer elements are pushed on the top of the stack. Pop is the reverse of push. Stacks made using arrays usually follows first-come first-leave fashion through **push**() and **pop**() operations.

Push – Stack operations 01/02

```
push(item)
  1. if top = max then
       Write " Overflow "
     else
       top ← top+1
       a[top] ← item;
     end if
```

```
void push(int item) {
    if(top == max) {
        printf("Overflow");
    } else {
        top++;
        a[top] = item;
    }
}
```

Pop – Stack operations 02/02

```
pop()
  1. item ← 0
  2. if top = NULL then
       Write " Underflow "
     else
       Item ← a[top]
     end if
  3. top ← top-1
```

```
void pop() {
    int item;
    if(top==NULL) {
        printf("Underflow");
    } else {
        item = a[top];
    }
    top--;
}
```

Traversing – Stacks basics

```
display()

  1. i ← 0

  2. Repeat until i <= top
      A. Write a[i]
      B. i ← i+1
     end repeat
```

```
void display() {

    int i;

    for(i=1;i<=top; printf("\n%d",a[i]), i++);

}
```

Queue structures – Data structures 04/05

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the REAR(also called tail), and the removal of existing element takes place from the other end called as FRONT(also called head).

Structure and Operations – Queues

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.

Enqueue – Queue operations 01/02

```
enqueue(item)

  1. if r = n then Write " Overflow "

  2. else r ← r+1

  3. a[r] ← item
```

```
void enqueue(int item) {

    if(r==n) printf("\nOverflow");

    else r++;

    a[r]=item;

}
```

Dequeue – Queue operations 02/02

```
dequeue()
  if r = 0 then
    Write " Underflow "
  else if r = f then
    f ← 1
    r ← 0
  else
    f ← f+1
  end if
```

```
void dequeue() {
    if(r==0) {
        printf("\nUnderflow");
    } else if(r==f) {
        f=1;r=0;
    } else {
        f++;
    }
}
```

Traversing – Queue basics

```
display()

  i ← f

  Repeat until i <= r
    Write a[i]
    i ← i+1
  End repeat
```

```
void display() {

    int i;

    for(i=f;i<=r; printf("\n%d",a[i]), i++)

}
```

Circular Queues – Data structures 04/05

Circular Queue is also 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.

Enqueue – Queue operations 01/02

```
enqueue(item)
        1. if f = 1 and r= n then Write " Overflow "
        2. if f = 0 then
             f ← 1 r ← 1
           else if r = n then
             r ← 1
           else
             r <-r+1;
             a[r] <-item;
           end if
```

```
void enqueue(int item) {
       if(f == 1 && r == n) {
        printf("\nOverflow");
    }

       if(f==0) {
        f = 1;
               r = 1;
    } else if(r == n) {
        r = 1;
    } else {
        r = r+1;
           a[r] = item;
    }

}
```

Dequeue – Queue operations 02/02

```
dequeue(item)

  1. item <-0

  2. if f = 0 then Write " Underflow "

        item <-a[f]

      end if

  3. if f = r then
       r <-0
       f <-0
     else if f = n then
       f <-1
     else
       f <-f+1
     end if
```

```
void dequeue() {

    int item;

    if(f==0) printf("\nUnderflow");

       item=a[f];

    if(f==r) {
       r=0;
       f=0;
    } else if(f==n) {
       f=1;
    } else {
       f=f+1;
    }

}
```