```
struct node {
            int info;
     struct node *link;
};

struct  node * start=null ;     //thepointer variable start will be  declared as
 p=start;                  //intially structer  ptr assigned to start value

p->info=start->info;
p=prt->link;
```

1) Traverse a linked list

```
void display(struct node *start)
{
struct node *p;
if(start=null)
{
printf("list is empty");
return;
}
p=start;
while(ptr!=NULL)
{
printf("%d",ptr->info);
p=p->link;
}
```

2) count num of elements in a linked list

```
void count(struct node *start)
{
struct node *p;
int count=0;
p=start;
while(ptr!=NULL)
{
p=p->link;
count++;
}
printf("no of elements are %d",count);
}
```

2) searching in a linked list

```
void search(struct node *start, int item)
{
struct node *p=start;
int pos=1;
while(p!=NULL)
{
if(p->info==item)    //search sucessful
{
printf("item %d  pos %d",item,pos);
return;
}
      p=p->link;
pos++;
}
```

4)insert in a linked list

```
tmp=(struct node*)malloc(sizeof(struct node));
tmp->>info=data  ;              //data mean inserting elemnt info
```

insertion at begining of list

```
struct node *addatbeg(struct node *start, int data)
{
struct node *tmp;
tmp=(struct node*)malloc(sizeof(struct node));
tmp->info=data  ;
tmp->link=start  ;
start=tmp;

}
```

insertion in empty list

```
struct node *addatempty(struct node *start, int data)
{
struct node *tmp;
tmp=(struct node*)malloc(sizeof(struct node));
tmp->link=null ;
start=tmp;

}
```

insertion at the end of list

```
struct node *addatend(struct node *start, int data)
{
struct node *p,*tmp;
tmp=(struct node*)malloc(sizeof(struct node));
tmp->info=data  ;
p=start;
while(p->link!=NULL)
p=p->link;

p->link=tmp  ;
tmp->link=null;
return start;

}
```
insertion in b/w the list nodes   *//p,q are two nodes*
```
tmp->link=p->link;
p->link=tmp;
```

insertion after a node

```
struct node *addatafter(struct node *start, int data, int item)
{
struct node *tmp,*p;
p=start;
while(p!=null)
{
if(p->info==item)
{
tmp=(struct node*)malloc(sizeof(struct node));
tmp->info=data  ;
tmp->link=p->link;
p->link=tmp;
return start;
}
p=p->link;
}
printf(%d not present in list ,item);
return start;
}
```

insertion before node

```
struct node *add before(struct node *start, int data, int item)
{
struct node *tmp,*p;
if(start==null)
{
prinf("list is empty");
return start;
}
////if data inserted before first node////////////////
if(item==start->info)
{
tmp=(struct node*)malloc(sizeof(struct node));
tmp->info=data  ;
tmp->link=start  ;
start=tmp;
return start;
}

p=start;
while(p->link!=null)
{
if(p->link->info==item)
{
tmp=(struct node*)malloc(sizeof(struct node));
tmp->info=data  ;
tmp->link=p->link;
p->link=tmp;
return start;
}
p=p->link;
}
printf(%d not present in list ,item);
return start;
}
```

insertion at given pos

```
struct node *addatpos(struct node *start, int data, int pos)
{
struct node *tmp,*p;
int i;
p=start;
for(i=0;i<pos-1&&p!=null;i++)
p=p->link;
if(p==null)
printf("there are less than %d elements,pos");
else
{
tmp=(struct node*)malloc(sizeof(struct node));
tmp->info=data  ;
if(pos==1)
{
tmp->link=start;
start=tmp;
}
else
{
tmp->link=p->link;
p->link=tmp;
}
}
return start;
```

create a linked list

```
struct node *create_list (struct node *start)
{
int i,n,data;
printf("enter the no of nodes");
scanf("%d",&n);
start=null;
if(n==0)
return start;
printf("enter the element to be inserted:");
scanf("%d",&data);
start=addatbeg(start,data)
if(i=2;i<=n;i++)
{
printf("enter elements to be inserted:");
scanf("%d",&data);
start=addatend(start,data);
}
return start;
```

deletion in a linked list

delete first node

```
struct node *delfirst (struct node *start)
{
struct node *tmp,;
tmp=(struct node*)malloc(sizeof(struct node));
temp=start;
start=start->link;
free(tmp);
}
```

deletion of only node

only one node in list delete that

```
struct node *delfirst (struct node *start)
{
struct node *tmp,;
tmp=(struct node*)malloc(sizeof(struct node));
tmp=start;
start=null;
free(tmp);
}
```

deletion in b/w the list nodes

```
struct node *delfirst (struct node *start,int data)
{
struct node *tmp,*p;
tmp=(struct node*)malloc(sizeof(struct node));
p=start;
while(p->link!=null)
{
   if(p->link->info==data)
  {
   tmp=p->link;
   p->link=tmp->link;
   free(tmp);
   return start;
  }
p=p->link;
}
```

deletion at end of list

```
struct node *delfirst (struct node *start,int data)
{
struct node *tmp,*p;
tmp=(struct node*)malloc(sizeof(struct node));
if(start==null)
{
printf("list iis empty");
return start;
}
if(start->info==data)  //deletion of first node
{
temp=start;
start=start->link;
free(tmp);
return start;
}

p=start;                                        //deletion in b/w or at the end
while(p->link!=null)
{
   if(p->link->info==data)
  {
```

```
  tmp=p->link;
  p->link=tmp->link;
  free(tmp);
  return start;
 }
p=p->link;
}
printf("element not found%d",data);
return start;
}
```

reverse linked list

```
struct node *reverse(struct node *start,)
struct node prev,*ptr,*next ;                  //we will take three pointers
prev=null;                                    //intially
ptr=start;                                    //intially

while(ptr!=null)
{
next=ptr->link;
ptr->link=prev;
prev=ptr;
ptr=next;
}
start=prev;
return start;
}
```

Stack

Stack is a specialized data storage structure (Abstract data type).

It has two main functions **push** and **pop.**

Insertion in a stack is done using **push** function and

removal from a stack is done using **pop** function.

Stack allows access to only the last element inserted hence, an item can be inserted or removed from the stack from one end called the top of the stack.

, also called **Last-In-First-Out (LIFO)** list.

Stack has three properties: capacity stands for the maximum number of elements stack can hold, size stands for the current size of the stack and elements is the array of elements.

Singe Linked list

Singly linked list is the most basic linked data structure.
In this the elements can be placed anywhere in the heap memory which uses contiguous locations.

Nodes in a linked list are linked together using a next field, which stores the address of the next node in the next field of the previous node .
each node of the list refers to its successor and the last node contains the NULL reference. It has a dynamic size, which can be determined only at run time.

Basic operations of a singly-linked list are:

Insert – Inserts a new element at the end of the list.

Delete – Deletes any node from the list.

Find – Finds any node in the list.

Print – Prints the list.

Performance

1. The advantage of a singly linked list is its ability to expand to accept virtually unlimited number of nodes in a fragmented memory environment.
2. The disadvantage is its speed. Operations in a singly-linked list are slow as it uses sequential search to locate a node.

Doubly Linked List

Doubly-linked list is a more sophisticated form of linked list data structure.
Each node of the list contain two references (or links) – one to the previous node and other to the next node.
The previous link of the first node and the next link of the last node points to NULL.
In comparison to singly-linked list, doubly-linked list requires handling of more pointers but less information is required as one can use the previous links to observe the preceding element. It has a dynamic size, which can be determined only at run time.

Performance

1. The advantage of a doubly linked list is that we don't need to keep track of the previous node for traversal or no need of traversing the whole list for finding the previous node.
2. The disadvantage is that more pointers needs to be handled and more links need to updated.

Circular Linked List

Circular linked list is a more complicated linked data structure.
In this the elements can be placed anywhere in the heap memory unlike array which uses contiguous locations.
Nodes in a linked list are linked together using a next field, which stores the address

of the next node in the next field of the previous node i.e. each node of the list refers to its successor and the last node points back to the first node unlike singly linked list. It has a dynamic size, which can be determined only at run time.

Basic Operations on a Circular Linked List

Insert – Inserts a new element at the end of the list.
Delete – Deletes any node from the list.
Find – Finds any node in the list.
Print – Prints the list.

Performance

1. The advantage is that we no longer need both a head and tail variable to keep track of the list. Even if only a single variable is used, both the first and the last list elements can be found in constant time. Also, for implementing queues we will only need one pointer namely tail, to locate both head and tail.
2. The disadvantage is that the algorithms have become more complicated.

Related Tutorials :

Single Linked List	A self referential data structure. A list of elements, with a head and a tail; each element points to another of its own kind.	Double Linked List	A self referential data structure. A list of elements, with a head and a tail; each element points to another of its own kind in front of it, as well as another of its own kind, which happens to be behind it in the sequence.	Circular Linked List	Linked list with no head and tail - elements point to each other in a circular fashion.

Basic operations of a singly-linked list are:

Insert – Inserts a new element at the end of the list.
Delete – Deletes any node from the list.
Find – Finds any node in the list.
Print – Prints the list.

Queue

Queue is a specialized data storage structure (Abstract data type). . It has two main *operations enqueue and dequeue. Insertion in a queue is done using enqueue function and removal from a queue is done using dequeue function. An item can be inserted at the end ('rear') of the queue and removed from the front ('front') of the queue. It is therefore, also called First-In-First-Out (FIFO) list.*

Queue has five properties - capacity stands for the maximum number of elements Queue can hold, size stands for the current size of the Queue, elements is the array of elements, front is the index of first element (the index at which we remove the element) and rear is the index of last element (the index at which we insert the element).

Related Tutorials :

Stacks	Last In First Out data structures (LIFO). Like a stack of cards from which you pick up the one on the top (which is the last one to be placed on top of the stack). Documentation of the various operations and the stages a stack passes through when elements are inserted or deleted. C program to help you get an idea of how a stack is implemented in code.	Queues	First in First Out data structure (FIFO). Like people waiting to buy tickets in a queue - the first one to stand in the queue, gets the ticket first and gets to leave the queue first. Documentation of the various operations and the stages a queue passes through as elements are inserted or deleted. C Program source code to help you get an idea of how a queue is implemented in code.

/* Function to get the middle of the linked list*/

```
void printMiddle(struct node *head)
{
    struct node *slow_ptr = head;
    struct node *fast_ptr = head;

    if (head!=NULL)
    {
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;
            slow_ptr = slow_ptr->next;
        }
        printf("The middle element is [%d]\n\n", slow_ptr->data);
    }
}
```