LINKED LISTS

MODULE 3

Advantages of using arrays

- Arrays are simple to understand and use
- Data accessing is faster
  - Data can be accessed very efficiently just by specifying the array name and the index of the item (A[i])
  - Time required to access A[0] is same as A[1000]

Disadvantages of arrays

- The size of the array is fixed
  - Fixed amount of memory allocated before the start of execution.
  - Most of the time we cannot predict the memory space
  - Less memory is allocated and more memory is required during the execution.
- Array items are stored contiguously
  - Some times continuous memory location may not be available
- Insertion and deletion operations involving array is tedious job
  - If an array consists of more than 100 elements and we want to insert an element at ith position, then all the elements need to move one position to provide the space in the memory. Similarly for delete operation.

MEMORY MANAGEMENT (STATIC & DYNAMIC)

Static memory management

- The required memory is allocated to the variables at the beginning of the program. Memory to be allocated is fixed and is determined by the compiler at the compile time itself.
- Issues with static : array size with given allocation
  - can be large allocation for small group of data(here , there is wastage of memory that is allocated, that can't be used for any other purpose) or
  - could be less for large group of data

Dynamic memory allocation

- makes efficient use of memory by allocating the required amt of memory whenever needed.
- In most of the real time problems, can’t predict the memory requirements.
- Dynamic memory allocation does the job at run time.
- C provides the following dynamic allocation and de-allocation functions :
- (*i*) malloc( )
- (*ii*) calloc( )
- (*iii*) realloc( )
- (*iv*) free( )

- malloc() function allocates a block of contiguous bytes.
- The allocation can fail if the space in the heap is not sufficient to satisfy the request.
- If it fails, it returns a NULL pointer.
- So it is always better to check whether the memory allocation is successful or not before we use the newly allocated memory pointer.
- malloc.c
- calloc.c

- The calloc() function works exactly similar to malloc() function except for the fact that it needs two arguments as against one argument required by malloc() function.
- While malloc() function allocates a single block of memory space, calloc() function allocates multiple blocks of memory, each of the same size, and then sets all bytes to zero.
- The general form of calloc() function is
- ptr = (int_type*) calloc(n sizeof (block_size));
- ptr = (int_type*) malloc(n* (sizeof (block_size));

- In some situations, the previously allocated memory may be insufficient to run the correct application,
- *May want to* increase the memory space or want to reduce the memory space.
- In both the cases we want to change the size of the allocated memory block and this can be done by realloc() function.
- ptr = realloc(ptr, New_Size)
- Where 'ptr' is a pointer holding the starting address of the allocated memory block. And New_Size is the size in bytes that the system is going to reallocate
- In c++, we have new and delete operators, and also supports malloc(), calloc() etc

Singly linked list as data str(getnode and freenode)

Linked List

- A linked list is a non-sequential collection of data items.
- For every data item in the linked list, there is an associated pointer that would give the memory location of the next data item in the list.
- The data items in the linked list are not in consecutive memory locations.
- Accessing of these items is easier as each data item contained within itself the address of next data item.

Header nodes

- Extra node which does not represent any item in the list is called header node .
- A header node can be an empty node
- Stores the total number of nodes in the list
- Be just an single node
- Have just a pointer pointing to the last node

Array implementation of lists

Array implementation has different operations

- Insert a node
- Delete a node
- getnode
- freenode
- Insert after
- Delete after

Advantage & Disadvantages

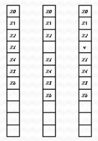
- Linked lists are dynamic data structures: grow or shrink
- Efficient memory utilization: memory is allocated when required.
- Insertion and deletions are easier & efficient: provide flexibility in inserting a data item at specified position & deletion of a data item.

- More Memory: if the number of fields are more.
- Access to an arbitrary data item is number some & time consuming.

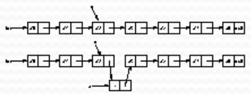
Linked List as a Data Structure

- An item is accesses in a linked list by traversing the list from its beginning.
- An array implementation allows acccess to the *n*th item in a group using single operation, whereas a list implementation requires *n* operations.
- The advantage of a list over an array occurs when it is necessary to insert or delete an element in the middle of a group of other elements.

Element *x* is inserted between the third an fourth elements in an array



Inserting an item *x* into a list after a node pointed to by *p*

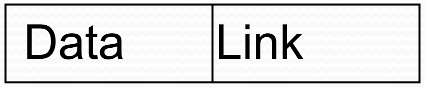


Components of Linked List

- Linked list is a non-sequential collection of a data items called nodes.
- Each node in a linked list contains two fields.
  - Data field
  - Link field
- Data field contains an actual value to be stored.
- Link field contains the address of the next data item(node).
- The address used to access a particular node is a pointer.
- The link field of the last node contains zero rather than a valid address.
- It is a null pointer & indicates the end of the list.
- https://examradar.com/single-linked-list/

Representation of a linked list

- Physical View:



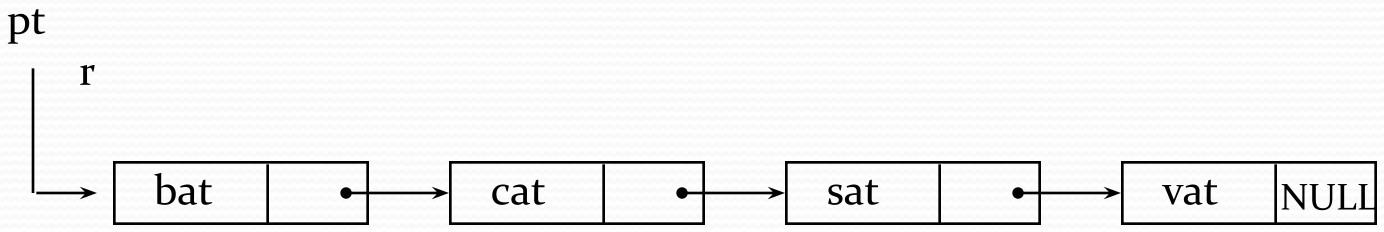
- Logical View:

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

- The first one is an integer data item & second one is link   pointer to the next node of same type.
- Such structures are called self-referential structures.

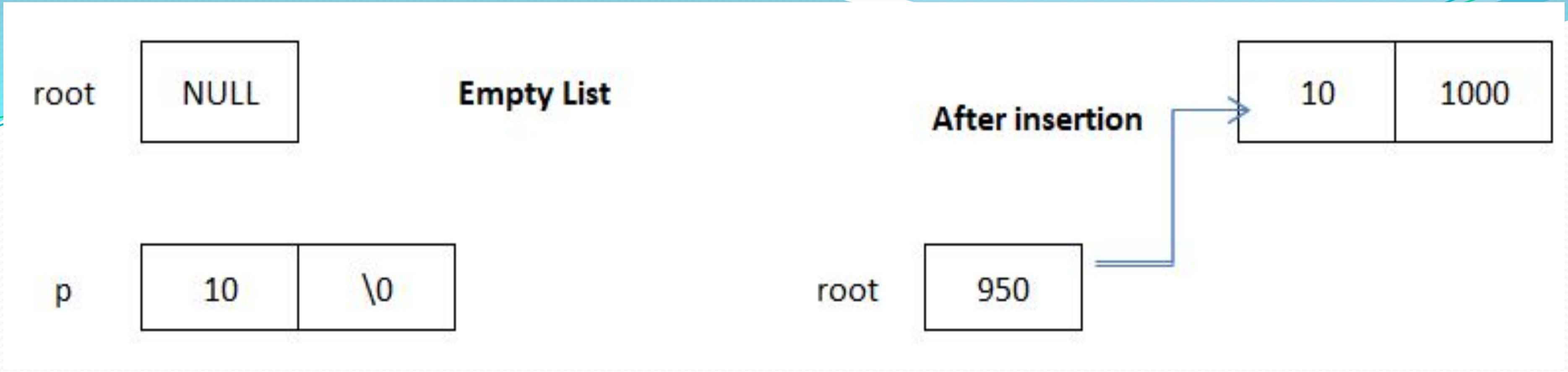
Singly Linked Lists

- **compose of data part and link part**
- **link part contains address of the next element in a list**
- **non-sequential representations**
- **size of the list is not predefined**
- **dynamic storage allocation and deallocation**



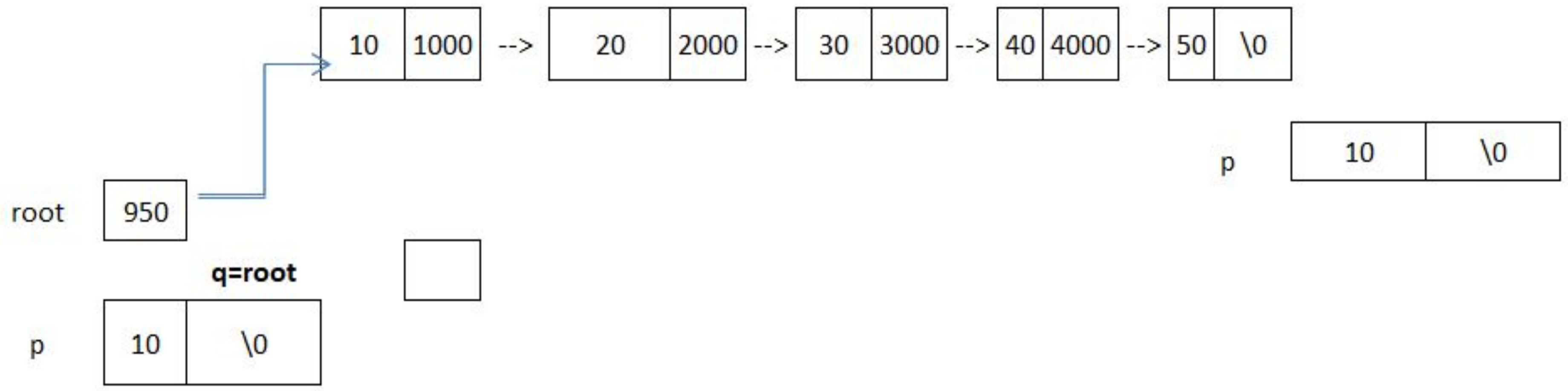
- Start pointer points to first node
- Last node in list points to null
- Basic operations :
  - Create,
  - Insert(at beginning of list, middle or end of list)
  - Delete(at beginning of list, middle or end of list)
  - Traverse
- Creation : creation of LL starts with creation of node. Sufficient memory needs to be allocated which is done using malloc(). Also done using getNode(), into which then data is stored.

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

struct node* root;
```

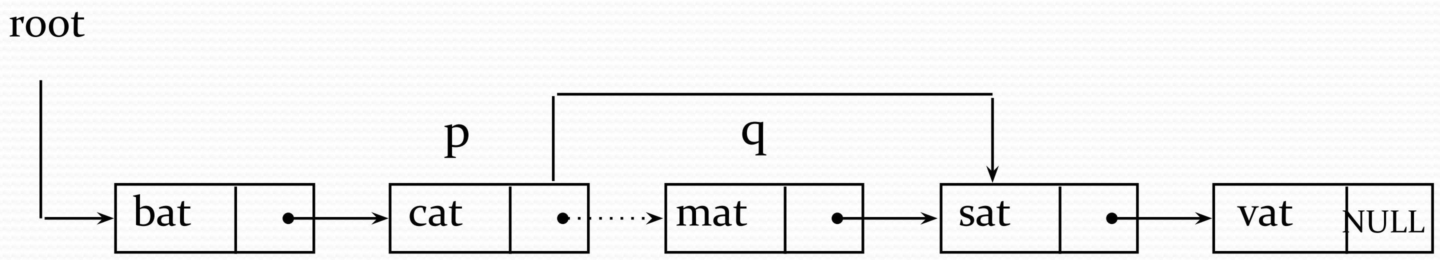


```
//Inserting New Node
struct node* p; //Create node
p= (struct node*)malloc(sizeof(struct node));
pf("Enter the data“)
scanf("%d", &p>data)
p>link=NULL

if(root == NULL) //Empty
   {
      root =p;
   }
```

```
//add at the begining
if(root==NULL)
{
      root=p;
}
else
{
      p->link=root//right
      root=p //left
}
```



```
else //Append end
   {
      struct node* q;
      q=root;
      while(q->link!=NULL)
      {
           q=q->link;
      }
      q->link=p;
}
```



```
//Delete first node
if(loc==1)
{
   p=root
   root=p->link
   p->link=NULL;
   free(p);
}
```

```
else
{
      struct node* p=root, *q;
      int i=1;
      while(i<loc-1) //specified node
      {
            p=p->link
            i++
      }
      q=p->link
      p->link=q->link
      q->link=NULL
      free(q)
}
```

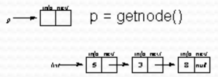
Some Notations for use in algorithm (Not in C programs)

- *p*: is a pointer
- *node(p)*: the node pointed to by *p*
- *info(p)*: the information portion of the node
- *next(p)*: the next address portion of the node
- *getnode()*: obtains an empty node
- *freenode(p)*: makes *node(p)* available for reuse even if the value of the pointer *p* is changed.

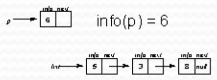
Inserting an item *x* into a list after a node pointed to by *p*

```
q=getnode();
info(q)=x;
next(q)=next(p);
next(p)=q;
```

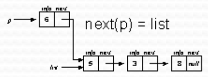
Adding an Element to the front of a Linked List



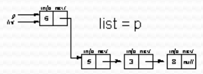
Adding an Element to the front of a Linked List



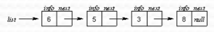
Adding an Element to the front of a Linked List



Adding an Element to the front of a Linked List

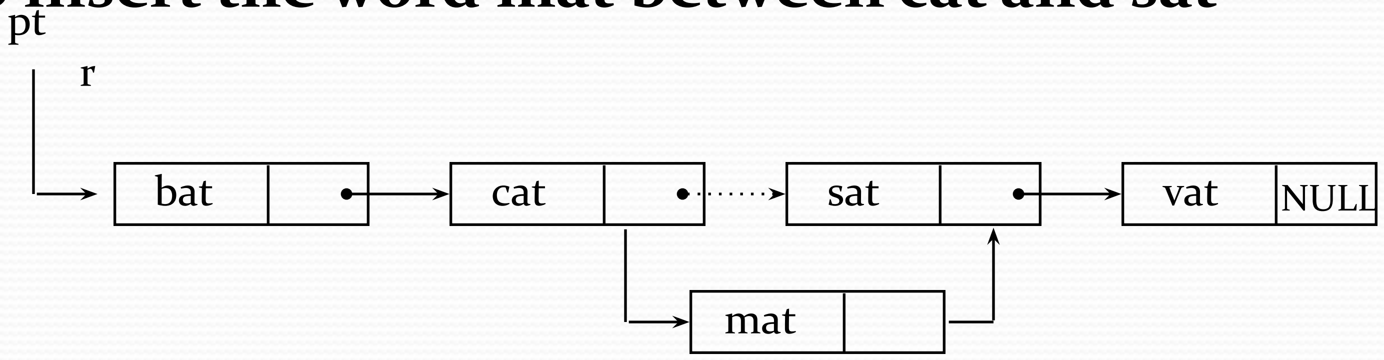


Adding an Element to the front of a Linked List



Singly Linked Lists

- **To insert the word mat between cat and sat**

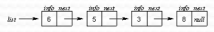


1) get a currently unused node (paddr)
2) set *paddr*'s data to mat
3) set *paddr*'s link to point to the address found in the link of the node cat
4) set the link of the node cat to point to *paddr*

Deleting an item *x* from a list after a node pointed to by *p*

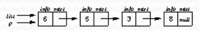
```
q=next(p);
x=info(q);
next(p)=next(q);
freenode(q);
```

Removing an Element from the front of a Linked List

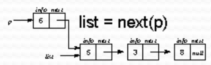


Removing an Element from the front of a Linked List

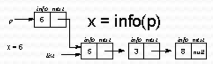
p = list



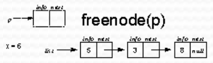
Removing an Element from the front of a Linked List



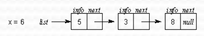
Removing an Element from the front of a Linked List



Removing an Element from the front of a Linked List

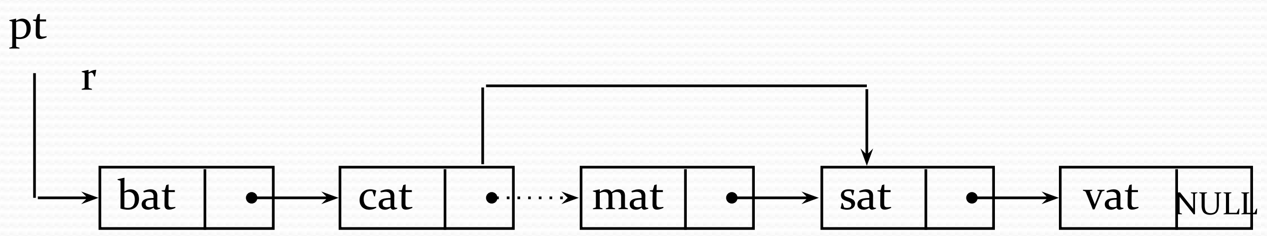


Removing an Element from the front of a Linked List



Singly Linked Lists

- **To delete mat from the list**



1) find the element that immediately precedes mat, which is cat

2) set its link to point to mat's link

cat->link=mat->link

free(mat);

Code for getnode and freeNode

```
NODEPTR p;
p = getnode();  //internally

NODEPTR getnode(void)
{
NODEPTR p;
p=(NODEPTR) malloc (sizeof (struct node));
return (p);
}
```

- Freenode(p);

```
 void freenode(NODEPTR p)
{
free(p);
}
```

Linked list implementation and stack implementation

- Header node : A *header node* is a special node that is found at the *beginning* of the list.
  - A list that contains this type of node, is called the header-linked list. This type of list is useful when information other than that found in each node is needed.
  - This special node is used to store number of nodes present in the linked list. In other linked list variant, if we want to know the size of the linked list we use traversal method.
  - But in Header linked list, the size of the linked list is stored in its header itself.

Linked List Implementation of Stacks – PUSH(S,X)

- The first node of the list is the *top* of the stack. If an external pointer *s* points to such a linked list, the operation *push(s,x)* may be implemented by

```
p=getnode();
info(p)=x;
next(p)=s;
s=p;
```

Linked List Implementation of Stacks – POP(S)

- The operation *x=pop(s)* removes the first node from a nonempty list and signals underflow if the list is empty:

```
if (empty(s))
{
/* checks whether s equals null */
printf('stack underflow');
exit(1);
}
Else
{
p =s;
s=next(p);
x = info(p);
freenode(p);
}
```

Doubly linked list

- A linear collection of nodes where each node is divided into three parts
- Info – holds the information
- llink - pointer contains the address of the left node or previous node in the list
- rlink – pointer which holds address of the right node or the next node in the list

Insert a node

```
 void insert(int item)
{
if(head==NULL)
head= (node*)malloc(sizeof(node));
head  -> data=item;
head ->prev = NULL;
head ->next = NULL;
}
else
{ struct node *temp;
temp=(node*)malloc (sizeof(node));
temp ->data=item;
temp ->prev=NULL; temp ->next = head;
head -> prev =temp; head=temp;
}
    }
```

```
insertatend(int item)
{
if( head == NULL)
head=(node *)malloc (size of(node));
head -> data = item;  head ->prev= NULL;
head ->next = NULL;
}
else {
struct node *temp= head;
while(temp -> next ! = NULL)
{
temp = temp -> next;
}
struct node *temp2 = ((node *)malloc(size of (node));
temp -> next = temp2;
temp2 -> prev = temp;
temp2 -> next = NULL;   temp2 -> data = item;
}
}
```

Delete a node from front

```
void remove( int item)
{
struct node *temp=head;
head = head -> next;
head -> prev = null;
temp ->next = temp -> prev = null;
temp -> data = item;
free ( temp);
```

Circular lists

- A list where the next field in the last node contains a pointer back to the first node rather than the null pointer.