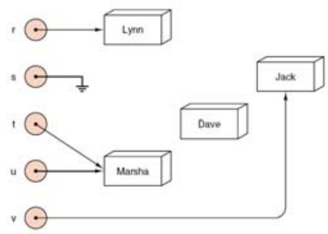
4: Linked Lists

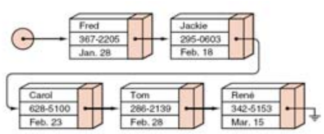
Pointers: An object, often a variable, that stores the location (that is the machine address) of some other object, typically of a structure containing data that we wish to manipulate. (Also sometimescalled a *link* or a *reference*)



1 1

Linked List

LIST: A list in which each entry contains a pointer giving the location of the next entry.



2

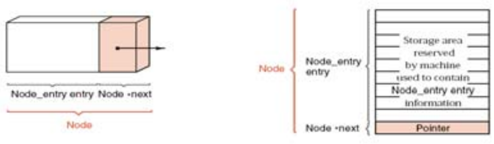
Basics of Linked Structures

- A linked structure is made up of nodes, each containing both the information that is to be stored as an entry of the structure and a pointer telling where to find the next node in the structure.
- We shall refer to these nodes making up a linked structure as the nodes of the structure, and the pointers we often call links.

3

```
struct Node {
  // data members
  Node_entry entry;
  Node *next;

  // constructors
   Node( );
   Node(Node_entry item, Node *add_on = NULL);
};
```



(a) Structure of a Node

(b) Machine storage representation of a Node

4

Node Constructor

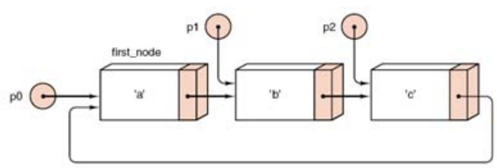
```
// first form of node constructor
Node :: Node( )
{
   next = NULL;
}

// second form of node constructor
Node :: Node(Node_entry item, Node *add_on)
{
   entry = item;
   next = add_on;
}
```

5

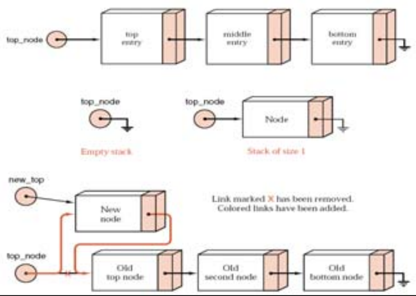
Example:

```
Node first_node(‘a’);    // Node rst node stores data ‘a’.
Node *p0 = &first_node;   // p0 points to first Node.
Node *p1 = new Node(‘b’); // A second node storing ‘b’ is created.
p0 -> next = p1;   // The second Node is linked after rst node.
Node *p2 = new Node(‘c’, p0); // A third Node storing ‘c’ is created.
                              // The third Node links back to the first_node, *p0.
p1->next = p2;   // The third Node is linked after the second Node.
p2->next = p0;
```



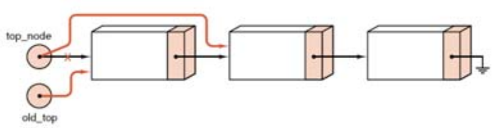
6

```
Error_code Stack :: push(const Stack_entry &item)  //push a stack
{
   Node *new_top = new Node(item, top_node);
   if (new_top == NULL) return overflow;
   top_node = new_top;
   return success;
}
```



7

```
Error_code Stack :: pop( )    // pop a stack
/* Post: The top of the Stack is removed. If the Stack is empty the method returns
    underow; otherwise it returns success. */
{
    Node *old_top = top_node;
    if (top_node == NULL)
          return underflow;
    top_node = old top->next;
    delete old_top;
    return success;
}
```



8

Linked List

```
//  Typical functions:

typedef struct node {
  int data; struct node *next; /* pointer to next
                                  element in list*/
} LLIST;

LLIST *list_add(LLIST **p, int i);
void list_remove(LLIST **p);
LLIST **list_search(LLIST **n, int i);
void list_print(LLIST *n);
```

9

add

```
LLIST *list_add(LLIST **p, int i)
{
    LLIST *n = new LLIST;
    if (n == NULL) return NULL;
    n->next = *p;  /* the previous element (*p) now
                      becomes the "next" element */
    *p = n; /* add new empty element to the front (head) of the list */
    n->data = i;
    return *p;
}
```

10

delete

```
void list_remove(LLIST **p) /* remove head */
{
    if (*p != NULL) {
        LLIST *n = *p;
        *p = (*p)->next;
        free(n); // delete n;
    }
}
```

11

search

```
LLIST **list_search(LLIST **n, int i)
{
        while (*n != NULL) {
                if ((*n)->data == i) {
                        return n;
                }
                n = &(*n)->next;
        }
        return NULL;
}
```

12

print

```
void list_print(LLIST *n)
{
   if (n == NULL) {
        cout << "list is empty\n");
   }
   while (n != NULL) {
        cout << n <<  n->next << n->data);
        n = n->next;
   }
}
```

13

main( )

```
int main( )
{
   LLIST *n = NULL;
   list_add(&n, 0); /* list: 0 */
   list_add(&n, 1); /* list: 1 0 */
   list_add(&n, 2); /* list: 2 1 0 */
   list_add(&n, 3); /* list: 3 2 1 0 */
   list_add(&n, 4); /* list: 4 3 2 1 0 */
   list_print(n);
   list_remove(&n); /* remove first (4) */
   list_remove(&n->next); /* remove new second (2) */
   list_remove(list_search(&n, 1)); /* remove cell containing 1 (first) */
   list_remove(&n->next); /* remove second to last node (0) */
   list_remove(&n); /* remove last (3) */
   list_print(n);
   return 0;
}
```

14

Linked List: Link list class

```
#include <iostream>

using namespace std;

class linklist
{
   private:

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

public:

      linklist();
      void append( int num );
      void add_as_first( int num );
      void addafter( int c, int num );
      void del( int num );
      void display();
      int count();
      ~linklist();
};
```

15

Linked List: Link list: empty. Only one pointer

```
linklist::linklist()
{
    p=NULL;
}
```

16

Linked List: Show list

```
void linklist::display()
{
    node *q;
    cout<<endl;

  for( q = p ; q != NULL ; q = q->link )
      cout<<endl<<q->data;

}
```

17

Linked List: append

```
void linklist::append(int num)
{
    node *q,*t;

  if( p == NULL )
  {
      p = new node;
     p->data = num;
     p->link = NULL;
  }
```

```
else
  {
     q = p;
     while( q->link != NULL )
        q = q->link;

     t = new node;
     t->data = num;
     t->link = NULL;
     q->link = t;
  }
}
```

18

Linked List: add_as_first

```
void linklist::add_as_first(int num)
{
   node *q;

   q = new node;
   q->data = num;
   q->link = p;
   p = q;
}
```

19

Linked List: Add-after

```
void linklist::addafter( int c, int num)
{
    node *q,*t;
  int i;
  for(i=0,q=p;i<c;i++)
  {
      q = q->link;
    if( q == NULL )
    {
    cout<<"\nThere are less than "
         <<c<<" elements.";
       return;
    }
  }
```

```
   t = new node;
   t->data = num;
   t->link = q->link;
   q->link = t;
} //end of function
```

20

Linked List: delete

```
void linklist::del( int num )
{
  node *q,*r;
  q = p;
  if( q->data == num )
  {
      p = q->link;
    delete q;
    return;
  }

  r = q;
```

```
while( q!=NULL )
  {
      if( q->data == num )
    {
        r->link = q->link;
      delete q;
      return;
    }

    r = q;
    q = q->link;
  }
  cout<<"\nElement "<<num<<" not Found.";
}
```

21

Linked List: count

```
int linklist::count()
{
   node *q;
   int c=0;
   for( q=p ; q != NULL ; q = q->link )
        c++;

   return c;
}
```

22

Linked List: Destructor ~

```
linklist::~linklist()
{
   node *q;
   if( p == NULL )
        return;

   while( p != NULL )
   {
      q = p->link;
      delete p;
      p = q;
   }
}
```

23

Linked List: main( )

```
int main()
{
   linklist ll;
   cout<<"No. of elements = "
        <<ll.count();
   ll.append(12);
   ll.append(13);
   ll.append(23);
   ll.append(43);
   ll.append(44);
   ll.append(50);

   ll.add_as_first(2);
   ll.add_as_first(1);
```

```
   ll.addafter(3,333);
   ll.addafter(6,666);

   ll.display();
   cout<<"\nNo. of elements = "
        <<ll.count();

   ll.del(333);
   ll.del(12);
   ll.del(98);
   cout<<"\nNo. of elements = "
        <<ll.count();
   return 0;
}
```

24

Linked Queues: Class declaration

```
class Queue {
public:                    // standard Queue methods
Queue( );
bool empty( ) const;
Error_code append(const Queue entry &item);
Error_code serve( );
Error_code retrieve(Queue entry &item) const;
// safety features for linked structures
Queue( );
Queue(const Queue &original);
void operator = (const Queue &original);
protected:
Node *front, *rear;
};
```

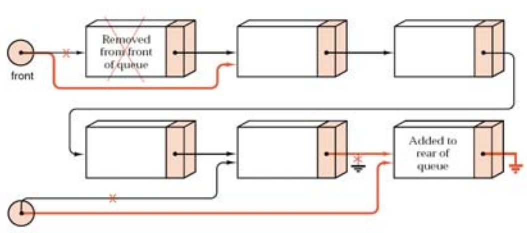
25

Linked Queues: Constructor

```
Queue :: Queue( )
/* Post: The Queue is initialized to be
   empty. */
{
    front = rear = NULL;
}
```

26

Linked Queues: Insertion and Deletion



27

Linked Queues: Insert (Append)

```
Error_code Queue :: append(const Queue entry &item)
/* Post: Add item to the rear of the Queue and return a
   code of success or return a code of overflow if dynamic
   memory is exhausted. */
{
   Node *new_rear = new Node(item);
   if (new_rear == NULL) return overflow;
   if (rear == NULL) front = rear = new_rear;
   else {
      rear->next = new_rear;
      rear = new_rear;
   }
   return success;
}
```

28

Linked Queues: Delete (served)

```
Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue
   is empty, return an Error code of underflow. */
{
   if (front == NULL) return underflow;
   Node *old_front = front;
   front = old_front->next;
   if (front == NULL) rear = NULL;
   delete old_front;
   return success;
}
```

29