DATA STRUCTURE

All Notes Are Verified by NPTL and Published By

WWW.ASKTOHOW.COM

IIT Guwahati

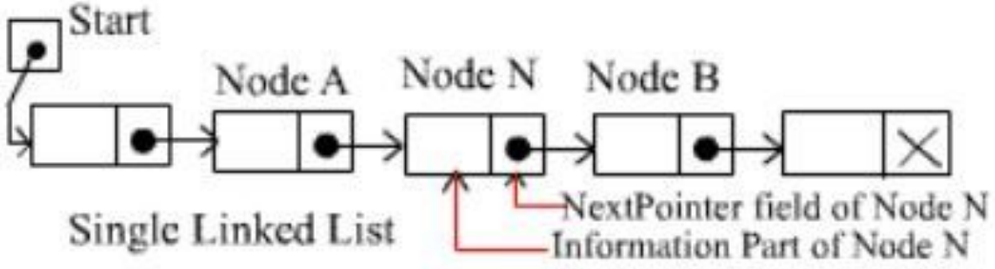
NPTL

C.E.O of ASKTOHOW

Deepak

What is are Linked Lists?

A linked list is simply a chain of structures which contain a pointer to the next element. It is dynamic in nature. Items may be added to it or deleted from it at will.



This definition applies only to Singly Linked Lists - Doubly Linked Lists and Circular Lists are different.

A list item has a pointer to the next element, or to 0 if the current element is the tail (end of the list). This pointer points to a structure of the same type as itself. This structure that contains elements and pointers to the next structure is called a Node.

Each node of the list has two elements:

- the item being stored in the list *and*
- a pointer to the next item in the list

Some common examples of a linked list:

- Hash tables use linked lists for collission resolution
- Any "File Requester" dialog uses a linked list
- Binary Trees
- Stacks and Queues can be implemented with a doubly linked list
- Relational Databases (e.g. Microsoft Access)

Algorithm for inserting a node to the List

- allocate space for a new node,
- copy the item into it,
- make the new node's next pointer point to the current head of the list and
- make the head of the list point to the newly allocated node.

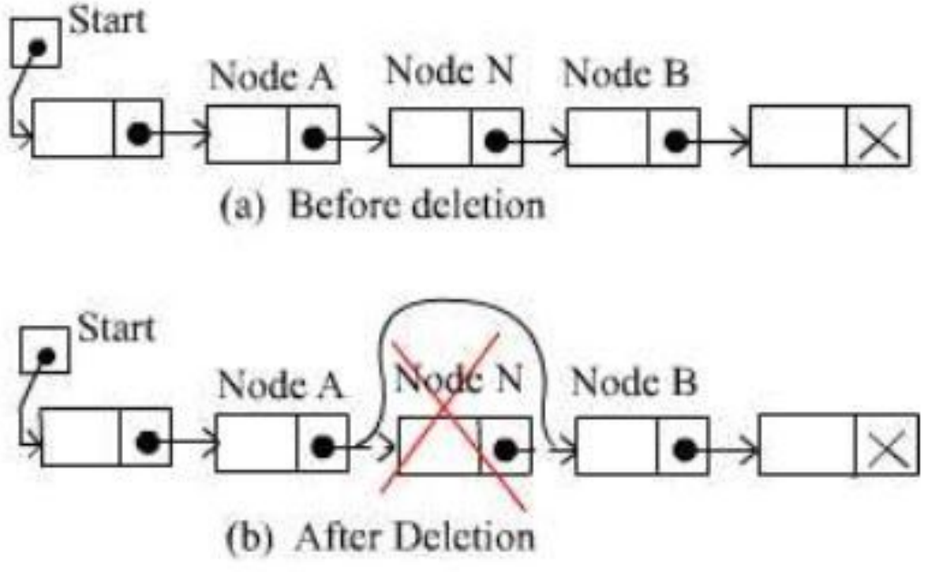
This strategy is fast and efficient, but each item is added to the head of the list. Below is given C code for inserting a node after a given node.

C Implementation

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

```
void insafter(int p, int x)

  {
int q;

   if(p == -1)

    {
printf("void insertion\n");
return;

    }
  q = getnode(); / * getnode() returns a pointer to newly allocated
node */

  node[q].info = x;

  node[q].next = node[p].next;

  node[p].next = q;

return;

  } /* end insafter
```

Algorithm for deleting a node from the List



Step-1:Take the value in the 'nodevalue' field of the TARGET node in any intermediate variable. Here node N.

Step-2: Make the previous node of TARGET to point to where TARGET is currently pointing

Step-3: The nextpointer field of Node N now points to Node B, Where Node N previously pointed.

Step-4: Return the value in that intermediate variable

There are also two special cases. If the deleted node N is the first node in the list, then the Start will point to node B; and if the deleted node N is the last node in the list, then Node A will contain the NULL pointer.

Below is a C code for deleting a node after a given node.

C Implementation

/* routine delafter(p,px), called by the statemant delafter(p,&x), deletes the node following node(p) and stores its contents in x */

```
void delafter(int p, int *px)
   {
  int q;
   if ((p == -1) || (node[p].next == -1))
    {
printf("void deletion\n");
return;
    }
  q=node[p].next;
  *px = node[q].info;
  node[p].next = node[q].next;
  freenode(q);
return;
  }

/* end delafter */
```

The operation of adding an element to the front of a linked list is quite similar to that of pushing an element on to a stack. A stack can be accessed only through its top element, and a list can be accessed only from the pointer to its first element. Similarly, removing the first element from a linked list is analogous to popping from a stack.

A linked-list is somewhat of a dynamic array that grows and shrinks as values are added to it and removed from it respectively. Rather than being stored in a continuous block of memory, the values in the dynamic array are linked together with pointers. Each element of a linked list is a structure that contains a value and a link to its neighbor. The link is basically a pointer to another structure that contains a value and another pointer to another structure, and so on. If an external pointer *p* points to such a linked list, the operation *push(p, t)* may be implemented by

```
f = getnode();
    info(f) = t;
    next(f) = p;
p = f;
```

The operation t = pop(p) removes the first node from a nonempty list and signals underflow if the list is empty

```
if(empty(p))
    {
printf('stack underflow');
exit(1);
   }
 else {
   f = p;
   p = next(f);
   t = info(f);
   freenode(f);
       }
```

GETNODE AND FREENODE OPERATIONS

The *getnode* operation may be regarded as a machine that manufactures nodes. Initially there exist a finite pool of empty nodes and it is impossible to use more than that number at a given instant . If it is desired to use more than that number over a given period of time, some nodes must be reused. The function of *freenode* is to make a node that is no longer being used in its current context available for reuse in a different context.

The list of available nodes is called the ***available list.*** When the available list is empty that is all nodes are currently in use and it is impossible to allocate any more, overflow occurs.

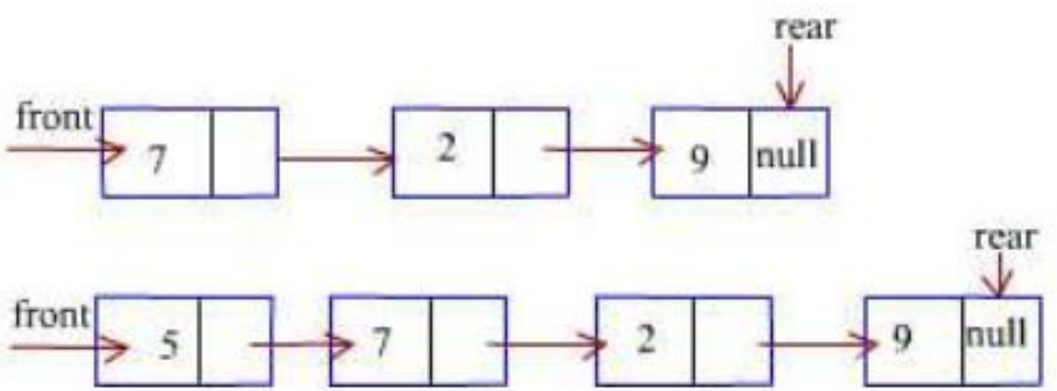
Assume that an external pointer *avail* points to a list of available nodes. Then the operation getnode and freenode are implemented as follows :

```
int getnode(void)
    {
int p;

    if (avail == -1)

      {

   printf("overflow\n");

   exit(1);

      }
    p = avail;

    avail = node[avail].next;

return(p);

   } /* end getnode */
```

```
void freenode (int p)
{
node[p].next = avail;
avail = p;
return;
} /* end freenode */
```

Queues can be implemented as linked lists. Linked list implementations of queues often require two pointers or references to links at the beginning and end of the list. Using a pair of pointers or references opens the code up to a variety of bugs especially when the last item on the queue is dequeued or when the first item is enqueued.

In a circular linked list representation of queues, ordinary 'for loops' and 'do while loops' do not suffice to traverse a loop because the link that starts the traversal is also the link that terminates the traversal. The empty queue has no links and this is not a circularly linked list. This is also a problem for the two pointers or references approach. If one link in the circularly linked queue is kept empty then traversal is simplified. The one empty link simplifies traversal since the traversal starts on the first link and ends on the empty one. Because there will always be at least one link on the queue (the empty one) the queue will always be a circularly linked list and no bugs will arise from the queue being intermittently circular. Let a pointer to the first element of a list represent the *front* of the queue. Another pointer to the last element of the list represents the *rear* of the queue as shown in fig. illustrates the same queue after a new item has been inserted.



Under the list representation, a queue *q* consists of a list and two pointers, *q.front* and *q.rear.* The operations are insertion and deletion. Special attention is required when the last element is removed from a queue. In that case , *q.rear* must also be set to *null*, Since in an empty queue both *r.front* and *q.rear* must be *null*.

The pseudo code for deletion is below:

```
if (empty(q))
  {
printf("Queue is Underflow");
exit(1);
  }
 f = q.front;
 t = info(f);
 q.front = next(f);
   if (q.front == null)
 q.rear = null;
 freenode(f);
return(t);
```

The operation insert algorithm is implemented

```
f = getnode();
 info(f) = x;
 next(f) = null;
     if (q.rear == null)
      q.front =f;
     else
    next(q.rear) = f;
    q.rear = f;
```

We can use a list to represent a priority queue in ordered list or unordered list. For an ascending Priority queue, insertion is implemented by the *place* operation, which keeps the list ordered, and deletion of the minimum element is implemented by the *delete* operation, which removes the first element from the list. A Descending priority queue can be implemented by keeping the list in decending order rather than ascending, or by using remove to delete the minimum element. In an ordered list, if you want to insert an element to the priority queue, it will require examining an average of approximately *n/2 nodes* but only one search for deletion.

An unordered list may also be used as a priority queue. If you want to insert an element to the list always requires examining only one node but always requires examining n elements for removal of an element.

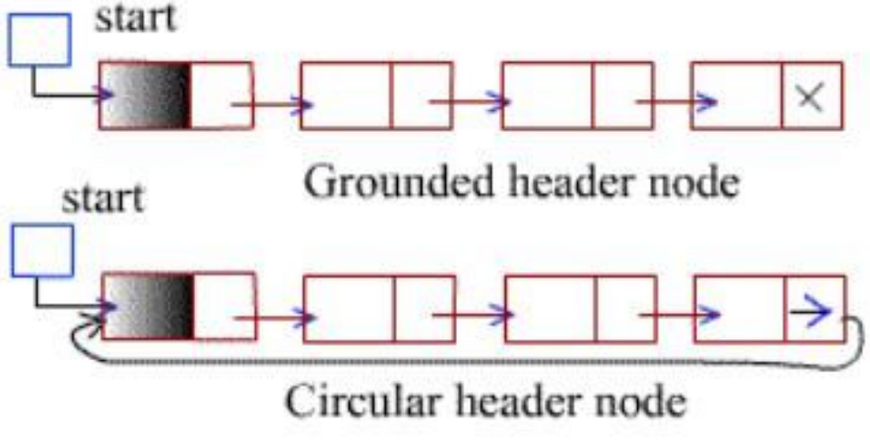
The advantage of a list over an array for implementing a priority queue is that an element can be inserted into a list without moving any other elements, where as this is impossible for an array unless extra space is left empty.

A header linked list is a linked list which always contains a special node called the *header node* at the beginning of the list. It is an extra node kept at the front of a list. Such a node does not represent an item in the list. The information portion might be unused. There are two types of header list

1. *Grounded header list* : is a header list where the last node contain the null pointer.
2. *Circular header list* : is a header list where the last node points back to the header node.

More often , the information portion of such a node could be used to keep global information about the entire list such as:

- number of nodes (not including the header) in the list
  count in the header node must be adjusted after adding or deleting the item from the list
- pointer to the last node in the list
  it simplifies the representation of a queue
- pointer to the current node in the list
  eliminates the need of a external pointer during traversal



CIRCULAR LIST

Circular lists are like singly linked lists, except that the last node contains a pointer back to the first node rather than the null pointer. From any point in such a list, it is possible to reach any other point in the list. If we begin at a given node and travel the entire list, we ultimately end up at the starting point.

Note that a circular list does not have a natural "first or "last" node. We must therefore, establish a first and last node by convention - let external pointer point to the last node, and the following node be the first node.

Stack as a Circular List

Queue as a Circular List

Stack as a Circular List

A circular list can be used to represent a stack.

The following is a C function to push an integer x onto the stack. It is called by push(&stack, x), where stack is a pointer to a circular list acting as a stack.

```
void push(NODEPTR *pstack, int x)
{

NODEPTR p;

p = getnode();

p->info = x;

if (*pstack == NULL)

*pstack = p;

else

p->next = (*pstack) -> next;
(*pstack) -> next = p;
} /*end push*/
```

Stack as a Circular List

The following is a C function to pop an integer from the stack. It is called by pop(&stack), where stack is a pointer to a circular list acting as a stack.

```
int pop(NODEPTR *pstack)
    {

int x;

NODEPTR p;

    if (*pstack == NULL)
      {
printf("Stack underflow\n");

exit(1);
       } /*end if*/
  p = (*pstack) -> next;
  x = p->info
      if (p == *pstack)

   /* only one node on the stack */
    *pstack = NULL;
      else
  (*pstack) -> next = p->next;
  frenode(p);
  return(x);
      } /*end pop */
```

Queue as a circular list

It is easier to represent a queue as a circular list than as a linear list. If considered as a linear list, a queue is specified by two pointers, one to the front of the list and other to its rear. However, by using a circular list, a queue may be specified by a single pointer q to that list. node(q) is the rear of the queue and the following node is its front.

Queue as a circular list

The following is a C function to insert an integer x into the queue and is called by insert(&q, x)

```
void insert(NODEPTR *pq, int x)
  {

NODEPTR p;

  p = getnode();
  p->info = x;
    if (*pq == NULL)
      *pq = p;

    else
 p->next = (*pq) -> next;
 (*pq) -> next = p;
 *pq = p;
return;
 } /*end insert*/
```

Queue as a circular list

To insert an element into the rear of a circular queue, the element is inserted into the front of the queue and the circular list pointer is then advanced one element, so that the new element becomes the rear.

Exercise:

Write an algorithm and a C routine to concatenate two circular lists.

Doubly Linked Lists

Doubly linked lists are like singly linked lists, in which for each node there are two pointers -- one to the next node, and one to the previous node. This makes life nice in many ways:

- You can traverse lists forward and backward.
- You can insert anywhere in a list easily. This includes inserting before a node, after a node, at the front of the list, and at the end of the list and
- You can delete nodes very easily.

Doubly linked lists may be either linear or circular and may or may not contain a header node

Problems :

1. What are the differences between a linked list and an array?
2. Write a small program to delete the last node of a given linked list.
3. Using the same code, convert the existing list into a circular linked list.
4. Using the data given in question 2,Write a small function to traverse the linkedlist only once and find out the middle element.
5. Given a singly linked list, write a small subroutine getNth() to get nth node of the linked list.
6. Write a small subroutine insertNth() to insert nth node in a singly linked list.
7. What is a queue and how can you implement queue using a linked list?
8. What is a stack and how can you implement stack using a linked list?
9. Given below is the subroutine for deletion of a node in a queue.

```
Delete (struct node *list, struct node *first, struct node *last){
Struct node *temp;
temp=first;
first=first->next;
first->prev=NULL;
free(first);
}
```

   There is an error in this code. Modify the code so that it deletes exactly the first node of the queue.

10. Given two singly linked lists, how do you append them?

11. What is the minimum number of queues needed to implement a stack? Write an algorithm to do so.