Lists

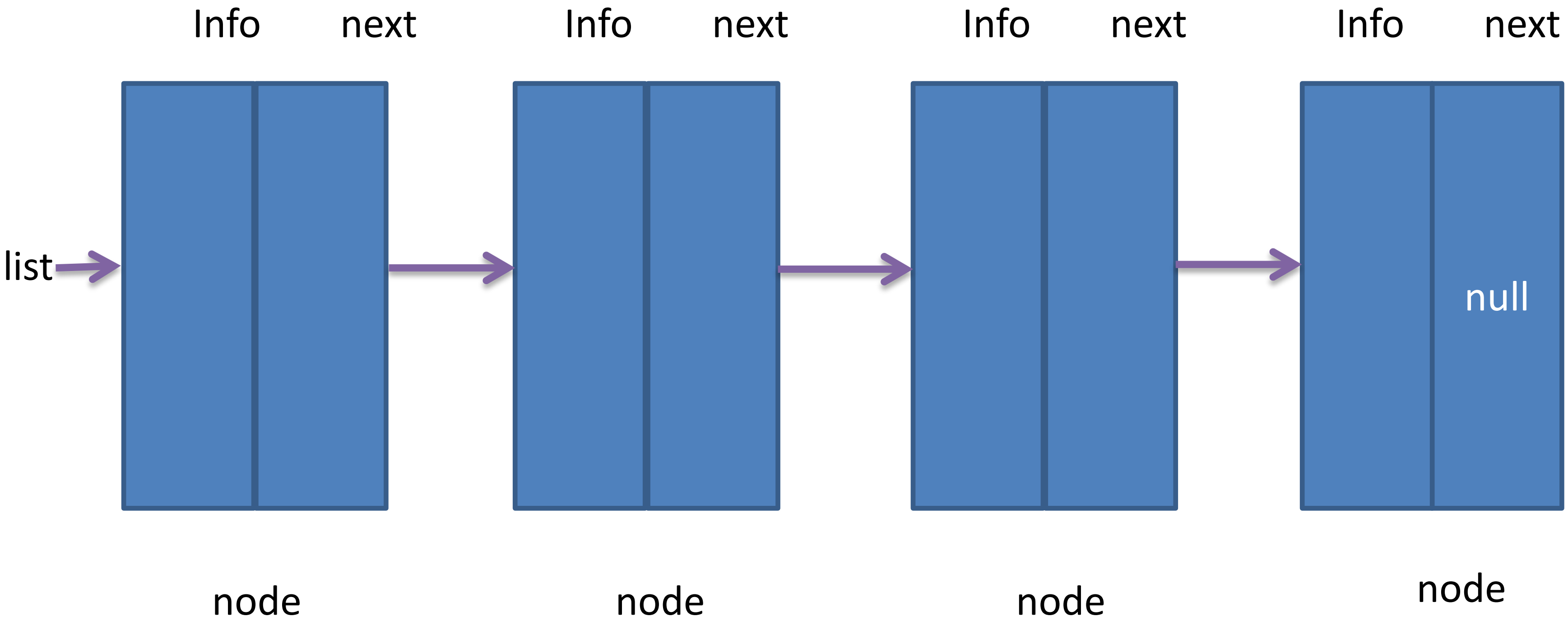
Unit 4

Introduction to Linked Linear Lists

- What are the **drawbacks of using sequential storage** to represent stacks and queues?
- One major drawback is that a **fixed amount of storage** remains allocated to the stack or queue even when the structure is actually using a smaller amount or possibly no storage at all.
- **In a sequential representation, the items of stack or queue are implicitly ordered** by the sequential order of storage.
- Thus if q items[x] represents an element of a queue.
- The next element will be q.items [x + 1] (or if x equals MAXQUEUE -1. q.items[0]).
- I**t provides relationship of physical adjacency**

Introduction

- ***linked linear list is a data structure which provides Explicit Ordering. (Logical adjacency)***
- Each item in the list is called a ***node*** and contains two fields, **an information field and a next** address field.
- **info -** The information field holds the actual elements on the list.
- **next -** The next address field contains the address of the next node in the list.
- Such an address, which is used to access a particular node, is known as a ***pointer.***



LINEAR LINKED LIST

Introduction

- The entire list is accessed from an ***external pointer*** – ***list (or First)*** that points to the ***first node*** in the list
    - list contains the address of the ***first node*** in the list.
- The ***next*** field of the ***last*** node in the list contains a special value known as ***null***.
- Null pointer signals the end of the list
- The list with no nodes is called an ***empty list*** or ***null list***.

Few Notations

- **If p is a pointer** to a node, **node(p)** refers to the node pointed to by p, **info(p)** refers to the information portion of that node, and **next(p)** refers to the next address portion and is therefore a pointer.
- Thus, if **next(p) is not null, info(next(p)) refers to the information portion** of the node that follows node(p) in the list.

Types of lists

- Types of lists
  - Singly Linked List
  - Doubly Linked List
  - Circular Linked List

Linked Linear Lists

- A list is a dynamic data structure.
- The **number of nodes on a list may vary dramatically** as elements are inserted and removed.
- The dynamic nature of a list may be contrasted with the static nature of an array, whose size remains constant.

Inserting Nodes into a List

- Obtains empty nodes and then adds them to the existing nodes.
- The mechanism for obtaining the empty nodes is ***p=getnode();***.
- It sets the contents of a variable named p to the address of that node.
- The value of p is then a pointer to this newly allocated node.
- The next step is to insert the integer 6 into info portion of the newly allocated node. This is done by operation
- info(p) = 6;

Inserting Nodes into a List

- After setting the info portion of node(p), it is necessary to set the next portion of that node.
- Since node(p) is to be inserted at the front of the list. this node should point to the current first node
- **next(p) = list;**
- This operation places the value of list into the next field of node(p).
- At this point, p points to list with the additional item included.
- The external pointer ***list*** should now point to the new additional node since it is at the beginning of the list.
- **list = p;**
- Which changes the value of list to the value of p.

Inserting Nodes into a List

- ***p*** is an auxiliary variable which is used during the process of modifying the list.
- The value of ***p*** is not necessary after the process of modifying. It can be reused.
- The pseudo code for this insertion in general is
  - ***p=getnode();***
  - ***info(p) =x;***
  - ***next(p)=list;***
  - ***list = p;***
    - where x is the value of the node to be inserted

Removing Nodes from a List

- The following operation are performed:

p = list

list = next(p);

x = info(p);

- The first node has been removed from list, and x has been set to the desired value.
- The variable p is used as an auxiliary variable during the process of removing the first node from the list.
- The list make no reference to p.
- **freenode(p);**

Removing Nodes from a List

Once this operation has been performed, it becomes illegal to reference node(p), since the node is no longer allocated.

- Since the value of p is a pointer to a node that has been freed, any reference to that value is also illegal.
- Another way of thinking of getnode and freenode is that getnode creates a new node whereas freenode destroys a node.

Representing Linked Lists in C

- Use the self referential structures to define a node's structure.
- To create new nodes when it is needed.
  – malloc function
- To remove the nodes that are no longer needed.
  – free function

Representing Linked Lists in C

- The general syntax is

```
struct node
{
      int info1, info2, info3……………;
      struct node * link;
} ;
typedef struct node NODE;
```

- Create a ***start*** pointer to the struct node called ***NODE***.
  - ***NODE * start;***

Create a Node

- Create a node dynamically using ***malloc ()***.
  - ***start= (NODE*) malloc (sizeof (NODE));***
    - This obtains a memory area that is sufficient to store the node and assign its address to the pointer variable ***start***.
    - This pointer indicates the beginning of the linked list.

Store data information

  - start→info = 100;
  - start→link = NULL;
    - The ***link*** part of the last node in a ***singly linked list*** should always contain ***NULL.***
- To traverse the list or to display the nodes, the ***start*** pointer should always point to the first node.
- Initially when the list is empty start = NULL;

Insert at the beginning of the List

Pseudo code:

- Step 1: Start
- Step 2 : [Create a new node that is to be inserted]
- newnode = getnode( );
- Step 3:[Assign the item into the info field of the newnode]
- Info [newnode]=ITEM
- Step 4:[Assign the link field of the newnode]
- link[newnode]=Start
  - [It makes a link from NEWNODE to the previous first node.

Insert at the beginning of the List

- Step 5 : Reassign ‘start’ with the NEWNODE so that a link is developed between start and newnode.
- start = newnode
- Step 6: Return

Delete at the beginning

Pseudo code:

- Step 1: Start
- Step 2: An auxiliary variable **temp** should point to start.
- **temp = start**
- Step 3: Assign item as the value of info field of first node
- **item= info[start] // or item = info(temp)**

Delete at the beginning

- Step 4: Reassign start with the next node so that the first node is removed.
- **start = link[start] // or start = link(temp)**
- Step 5: Free the node that is being pointed by temp
- **freenode(temp)**
- Step 6: Exit

GETNODE AND FREENODE OPERATIONS

- The getnode operation finds one new node from the pool of available nodes and makes it available to the algorithm.
- Thus each time that getnode is invoked, it presents its caller with a brand new node, different from all the nodes previously in use.
- 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.

GETNODE AND FREENODE OPERATIONS

- This available pool cannot be accessed by the programmer except through the getnode and freenode returns a node to the pool.
- It makes no difference which node is retrieved by getnode or where within the pool a node is placed by freenode.

GETNODE AND FREENODE OPERATIONS

- The available memory can be viewed as the list which is linked together by next field in each node
- The getnode operation removes the first node from this list and makes it available for use.
- The freenode operation adds a node to front of the list for reallocation by the next getnode.
- **The list of available nodes is called the available list.**

GETNODE AND FREENODE OPERATIONS

p = getnode(); can be implemented by

```
 if (avail == null) {
 printf(“overflow”);
 exit(1);
}
p = avail;
avail = next(avail);
```

GETNODE AND FREENODE OPERATIONS

- The implementation of freenode(p) is straightforward:

```
next(p) = avail;
avail = p;
```

LINKED IMPLEMENTATION of STACKS

- The operation of adding an element to the front of a linked list is quite similar to that of pushing an element onto a stack.
- A stack may be represented by a linear linked list.
- 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 IMPLEMENTATION of STACKS

- The operation empty(s) is merely a test of whether s equals null.
- The operation x = pop(s) removes the first node from a nonempty list and signals underflow if the list is empty.

```
 if (empty(s)) {
 printf('stack underflow');
 exit(1);
}
 else{
 p = s;
 s = next(p);
 x = info(p);
 freenode(p);
} /* end if */
```

Arrays v/s Linked lists

- The **disadvantages of representing a stack or queue by a linked list**
- A node in a linked list occupies **more storage** than a corresponding element in an array.
- Since two pieces of information per element are necessary in a list node.
- Only one piece of information is needed in the array implementation.
- An **array implementation allows access** to the nth item in a group using a **single operation**, whereas a list implementation requires n operations.
- It is necessary to pass through each of the first n-1 elements (Traversal)before reaching the nth element.

- **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.**

Suppose the item are stored as a list.

- If p points to an element of the list, inserting a new element after node(p) involves allocating a node, inserting the information, and adjusting two pointers.
- The amount of work required is independent of the size of the list.

Traversing a Linked List

Step 1: Start

Step 2: [Underflow condition]

if start = null then display linked list is empty and return

Step 3: Assign start to currptr // curr = start

Step 4: while curr!= null

process info[curr]

assign link[curr] to curr // curr = link(curr)

Step 5: Return

Displaying a Linked List

Step 1: Start

Step 2: [Underflow condition]

if start = null then display linked list is empty and return

Step 3: Assign start to curr //curr = start

Step 4: while curr!= null

display info[curr]

assign link[curr] to curr // curr = link(curr)

Step 5: Return

Insert at End

Step 1 : Start

Step 2 : Assign result of getnode() to newnode

Step 3: Assign item to info[newnode]

Step 4 : Assign null to link[newnode]

Step 5: [Empty List]

if start = null then Insert at beginning and return // start = newnode

Insert at End

Step 6: Assign start to curr

Step 7: [Traverse] while link[curr] != null

Assign link[curr] to curr

Step 8: Assign newnode to link[curr]

Step 9: Return

Insert at End

Pseudo Code:

```
p = getnode();
info(p) = x;
link(p) = NULL;
If (start == NULL)
    start = p, return
else
     curr = start;
     while (link(curr) != NULL)
            curr = link(curr)
     link(curr) = p
return
```

Delete from end of list

Step 1: Start

Step 2: [Empty List]

if start = NULL then display Empty List and return

Step 3: [One element in list]

if link[start] = null then

assign null to start and free node and return

// curr = start, start = NULL, free(curr)

Step 4: [More than one element in list]

Assign start to curr // curr = start

Delete from end of list

Step 5: Assign NULL to prev // prev = NULL

Step 6: while link[curr]!=NULL

Assign curr to prev // prev = curr

Assign link[curr] to curr // curr = link(curr)

Step 7: Assign NULL to link[prev] // link(prev) = NULL

Step 8: free(curr)

Step 9: Return

Insert a node at a particular position in the linked list

Step 1: Start
Step 2: read pos
Step 3: newnode = getnode()
Step 4: Assign item to info[newnode]
Step 5: if pos = 1 Insert at beginning
link(newnode) = start
start = newnode
return
Step 6: Assign start to curr // curr = start
Step 7: count = 1

Insert a node at a particular position in the linked list

Step 8: from beginning to pos –1 assign link[curr] to curr

while count < (pos-1)

curr = link(curr)

count ++

Step 9: If curr = null

display not possible to insert and return

Step 10: Assign link[curr] to link[newnode]

link(newnode) = link(curr)

Step 11: Assign newnode to link[curr]

link(curr) = newnode

Step 12: Return

Delete from a particular position

Step 1: Start

Step 2: Read pos

Step 3: If pos =1 then delete from beginning and return

// curr = start, start = link(start), free(curr)

Step 4: Assign start to curr // curr = start

Step 5: Assign NULL to prev // prev = NULL

Step 6: count = 1

Delete from a particular position

Step 7: from 1 to pos

Assign curr to prev, Assign link[curr] to curr

while count < pos

prev = curr

curr = link(curr)

count ++

Step 8: Assign link[curr] to link[prev]

link(prev) = link(curr)

Step 9: free curr

Step 10: return

Insertion after node p and deletion of a node after node p from a list

insafter(p, x)

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

delafter(p, x)

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

Assignment

- Delete all nodes whose info field is x
- Insert into an ordered linked list
- Count the number of nodes whose info field is >=60

Other Types of lists

- Lists with header nodes
- Noninteger lists

```
struct node {
        char info[100];
        struct node * next;
      };
```

```
struct node {
    char name[30];
    char id[10];
    char addr[100];
    float cgpa;
    char stream[20];
    struct node * next;
  };
```

CIRCULAR LINKED LIST

Introduction

- In a single linked list, the link part of the last node contains a ***NULL*** value.
- In a ***circular*** linked list, the link of the last node points to the first node of the linked list.
- **Advantage : Any node of the linked list can be accessed without going back and traversing again from the first node.**
- There is no first and last node in the linked list.
- A pointer **last** is considered during the processing of a circular linked list. (address of any one node usually the last node)
- Then next node is info(next(last))
- **If last == Null then empty list**

Create a CLL

- Step 1: Start
- Step 2: newnode = getnode()
- Step 3: assign item to info[newnode]
- Step 4: if last = NULL
    - assign newnode to last
    - assign last to link[newnode]
- Step 5: Return

Insert at Beginning - CLL

- Step 1: Start
- Step 2: newnode = getnode()
- Step 3: Assign item to info[newnode]
- Step 4: Assign link[last] to link[newnode]
- Step 5: Assign newnode to link[last]
- Step 6: Return

Insert at end- CLL – last node

- Step 1: Start
- Step 2: newnode = getnode()
- Step 3: assign item to info[newnode]
- Step 4: if last = null create_CLL and return
- Step 5: assign link[last] to link[newnode]
- Step 6: assign newnode to link[last]
- Step 7: assign newnode to last
- Step 8: Return

Delete from beginning - CLL

- Step 1: Start
- Step 2: if last = null
- display empty linked list and return
- Step 3: assign link[last] to curptr
- Step 4: assign link[curptr] to link[last]
- Step 5: Display info[curptr] and free curptr
- Step 6: Return

Delete from end - CLL

- Step 1: Start
- Step 2: if last = null display empty linked list and return
- Step 3: assign link[last] to curptr, last to prevptr
- Step 4: while [curptr]!=last
- assign curptr to prevptr
- assign link[curptr] to curptr
- Step 6: assign link[last] to link[prevptr]
- Step 7: Display info[curptr] and free curptr
- Step 8: Return

Home work

1. Count the nodes in a CLL
2. Insert at a specific position
3. Delete from a specific position

?``

DOUBLY LINKED LIST

Introduction

- In a SLL one could traverse only in one direction. To overcome this we can make use of DLL.
- A list that allows traversal in either the ***forward*** or ***backward*** direction is called ***doubly linked list***.
- This increases the performance and efficiency of the algorithms.

Introduction

- It contains two link fields –
  - ***prev***

    pointer that contains the address of the previous node
  - ***Next***

    Pointer that contains the address of the next node

PREV	INFO	NEXT

•The ***prev*** field of the ***first*** node and the ***next*** field of the ***last*** node always contains ***NULL***

DLL – Advantages / Disadvantages

- It can be traversed in either forward or backward direction.
- If a particular node's address is known, it is easy to know both the ***predecessor*** and the ***successor*** node address.
  - This makes it easy to insert and delete which is not possible in SLL
- It simplifies list management.
- But **Extra memory** is required by each node to store the ***prev*** pointer.
- Like SLL and CLL, here also we may have **header nodes** with global information about the list.

Representation of DLL in C

```
struct node
{
   int info1, info2, info3…………….;
   struct node * prev;
   struct node * next;
} ;
```

- typedef struct node *NODE;

Some Observations

- Note that if P is the address of a given node

then prev[p] – previous node of p

next[prev[p]] = p // node p itself

Similarly next[p] – next node of p

prev[next[p]] = p // node p itself

next[prev[p]] = p = prev[next[p]]

DLL - Creation

- Step 1: Start
- Step 2: newnode = getnode()
- Step 3: info[newnode] = item
- Step 4: start = newnode
- Step 5: prev[start] = null

  next[start] = null

NULL	ITEM	NULL

Insert at beginning

- Step 1: Start
- Step 2: newnode=getnode()
- Step 3: info[newnode] = item
- Step 4: prev[newnode] = null
- Step 5: next[newnode] = start
- Step 5: prev[start] = newnode
- Step 6: start = newnode
- Step 7: Return

DLL – Insert at End

- Step 1: Start
- Step 2: newnode = getnode()
- Step 3: info[newnode] = item
- Step 4: curr = start
- Step 5: If curr == null
- insert at beginning and return
- Step 5: While next[curr] != null
  curr =next[curr]
- Step 6: next[curr] = newnode
- Step 7: prev[newnode] = curr
- Step 8: next[newnode] = null
- Step 9: Return

DLL – Delete from Beginning

- Step 1: Start
- Step 2: curr = start
- Step 3: If curr == null
-     display empty list and return
- Step 4: Display info[curr]
- Step 5: If next[curr] == null
-         start = null
-     else
-         start = next[curr]
-         prev[start] = null
- Step 6: free curr
- Step 7: Return

DLL – Delete at End

- Step 1: Start
- Step 2: curr = start
- Step 3: If curr == null
- display empty list and return
- Step 4: if next[curr] == null   // one node
- display info[curr]
- start = null
- return
- Step 5: While next[curr] != null   // more than one node
- curr = next[curr]
- Step 6:next[prev[curr]] = null
- Step 7: Display info[curr]
- Step 8: Free curr
- Step 9: Return

DLL – Insert at a specific position

- Step 1: Start
- Step 2: newnode =getnode()
- Step 3: info[newnode] = item
- Step 4: if pos = 1
- insert at beginning and return
- Step 5: I = 1
- curr = start
- Step 6: while I is less than pos -1 and curr!=null
- curr = next[curr]
- increment I

- Step 7: If curr == null
-                     display invalid position and return
- Step 8: next[newnode] =next[curr]
- Step 9: prev[newnode] = curr
- Step 10: prev[next[curr]] = newnode
- Step 11: next[curr] = newnode
- Step 12: Return

DLL – Delete from a specific position

- Step 1: Start
- Step 2: curr = start
- Step 3: If curr == null
-                  display empty list and return
- Step 4: I = 1
- Step 5: while i<pos and curr != null
-                  curr = next[curr]

                   Increment I

- Step 6: next[prev[curr]] = next[curr]
- Step 7: prev[next[curr]] = prev[curr]
- Step 8: display info[curr]
- Step 9: free curr
- Step 10: Return

C routine for Deleting a node with address P

- This operation is not possible in SLL

```
 void delete(NODE * P, int * x)
{   NODE * l, *r;
    if (p == NULL)
      {  printf(“ deletion not possible\n”);
          return;
       }
    *x = p->info;
      l = p->prev;
      r = p->next;
      l->next = r;
      r->prev = l;
       free(p);
      return;
   }
```

Assignment

- Addition of two long integers using DLL

- Thank You