**Linked Lists**

* A linked list is a linear collection of data elements (called nodes), in which linear relation is given by links from one node to the next node.
* Similar to arrays, it is a linear collection of data elements of the same type.
* Different from arrays, adjacent data elements of a linked list are generally not stored adjacently in memory; instead they are dispersed.

**The node of a linked list**

* Linked list element (called node) use user defined structure type, typically contains two parts:
  + Data variables
  + Pointer to the next node, holding the addresses of next node

Example

Typedef struct node{

Int data;

Struct node \*next;

}NODE;

* A linked list can be accessed by a pointer, say start, pointing to the first node
* The last node will have no next node connected to it, it its next has a NULL value

**Linked list operations**

1. Create a node and linked list

* Creating a node can be done in two methods
  + Static methods use declaration to get memory block of a node
  + Dynamic method uses function malloc(sizeof(NODE)) to create a memory block of a node
  + The dynamic method is the default method for creating linked list nodes
* Creating a linked list is done by
  + Set a start node by a pointer to the first
  + Set links to connect the nodes
  + Se the NULL value to the next field of the last node

**Example**

Typedef struct node{

Int data;

Struct node \*next;

}NODE

NODE node1 = {1,NULL};

NODE node2 = {2,NULL};

NODE node3 = {3,NULL};

NODE \*start = &start1;

Node1.next = &node2;

Node2.next = &node3;

**Example 2**

NODE nodes[3] = {};

Nodes[0].data = 1;

Nodes[1].data = 2;

Nodes[2].data = 3;

NODE \*start = &nodes[0];

Nodes[0].next = &nodes[1];

Nodes[1].next = &nodes[2];

**Creating nodes by dynamic method**

NODE \*start;

NODE \*p1 = (NODE\*) malloc(sizeof(NODE));

P1->data=1;

P1->next = null;

Start = p1; //set start pointing to p1

NODE \*p2 = (NODE\*) malloc(sizeof(NODE));

P2->data = 2;

P2->next = NULL;

P1->next = p2; //link p1 to p2

**If the struct is declared as NODE p1, then you use dot notation. If NODE \*p1, then arrow notation**

**Traversing a linked list**

* From the start node, at each node, use next pointer to het the next node, until NULL pointer is reached

NODE \*p = start;

While(p!=NULL){

Printf(“%d”, p->data);

P = p->next;

}

**Search a linked list by key**

* Given a key value to search, returns the node address of the first matched node

Int key = value;

NODE \*p = start;

While(p!=NULL){

If(p->data ==key) break;

}

Return p;

**Insert at the beginning of linked list**

* Create a new node and insert it at b

NODE \*np = (NODE\*) malloc(sizeof(NODE));

Np->data = 4; //new value

Np->next = NULL;

Np->next = start; //start points to the first node

Start = np; //set the new start pointing to the new node

**Deleting the first node of a linked list**

NODE \*p = start;

Start = start ->next;

Free(p); //if node is dynamically created

**Singly Linked Lists**

* A singly linked list has the following properties:
  + The node contains data and one pointer to the next node of the same data type
  + Accessing by a pointer say start to the first node
  + The last node has next pointing to NULL

Create a new node

NODE \*new\_node(int val){

NODE \*p = (NODE\*)malloc(sizeof(NODE));

If(p!=NULL){

p->data = val;

p->next = NULL;

}

Return p;

}

NODE \*np = new\_node(1);

NODE \*start = np;

**Traversing singly linked lists**

Iterative algorithm: traverse a singly linked list

Input: start

Step 1: set ptr = start

Step 2: if ptr == NULL, goto step 5

Step 3: process ptr->data

Step 4: ptr = ptr->next, goto step 2

Step 5: stop

**Iterative algorithm, Time O(n), space: O(1)**

Void display(NODE \*start){

If(start==NULL) return;

NODE \*ptr = start;

While(ptr!=NULL){

Printf(“%d”, ptr->data);

Ptr = ptr->next;

}

}

**Recursive Algorithm, Time O(n), space: O(n) (not space efficient)**

Void display\_recursive(NODE \*start){

If (start == NULL) return;

Printf(“%d”, start->data);

Display\_recursive(start->next);

}

**Displaying backward**

Void display\_backward(NODE \*start){

If(start==NULL) return;

Display\_backward(start->next);

Printf(“%d”, start->data);

}

**Searching a Linked List**

NODE\* search(NODE \*ptr, int num){

While((ptr != NULL) && (ptr->data != num)){

Ptr = ptr->next;

}

Return ptr;

}

**Insert operation**

* Inserting a node into a linked list means adding a new node to the linked list. There are several variations depending on the position to insert.

**Insert at the beginning, time: O(1), space: O(1)**

NODE \*insert\_beg(NODE \*start, NODE \*np){

Np->next = start;

Start = np;

Return start;

}

**Insert at the end, time: O(n), space O(1)**

Void insert\_end(NODE \*\*startp, NODE \*np){

NODE \*ptr = \*startp;

If(ptr==NULL) \*startp = np;

While(ptr->next) ptr = ptr->next;

Ptr->next = np;

}

**Inserting at specific spot, time: O(n), space: O(1)**

Void insert\_after(NODE \*\*startp, NODE \*np, int key){

NODE \*ptr = \*startp;

While(ptr !=NULL && ptr->data !=key){

Ptr = ptr->next;

}

If (ptr!=NULL && ptr->data ==key){

Np->next = ptr->next;

Ptr->next = np;

}

}

**Deleting the first node of a linked list, time O(1), space: O(1)**

Void delete\_beg(NODE \*\*startp){

NODE \*ptr = \*startp;

If(ptr!=NULL){

\*startp = ptr->next;

Free(ptr); //free the memory block of the deleted node

}

}

**Deleting a specific node, time O(n), space O(1)**

Void delete(NODE \*\*startp, int value){

NODE \*ptr = \*startp;

NODE \*pre\_ptr = NULL;

While(ptr!=NULL && ptr->data !=value){

Pre\_ptr = ptr;

Ptr = ptr->next;

}

If(ptr==NULL) return;

Else if(pre\_ptr==NULL){ //first node is matched, delete the first node

\*startp = ptr->next;

Free(ptr);

}

Else{ // a non first node is match, delete it

Ptr\_ptr->next = ptr->next;

Free(ptr);

}

}

//call the function like this

Delete(&start, 2)

**Doubly linked lists**

* A doubly linked list has the following properties
  + The node structure consists of data variables and two pointers, next pointing to the next node, and prev pointing to the previous node
  + The first node has its prev pointing to NULL and the last node has its next pointing to NULL
  + There are two accessing pointers, start pointing to the first node and end pointing to the last node

Example: node structure of doubly linked list

Typedef struct node{

Struct node \*prev;

Int data;

Struct node \*next;

} NODE;

NODE \*new\_node(int val){

NODE \*np = (NODE\*) malloc (sizeof(NODE));

If (np==NULL){

Printf(“Malloc fails”); return NULL;

}

Np->data = val;

Np->prev = NULL;

Np->next = NULL;

Return np;

**Circular linked lists**

* They have the following properties:
  + The node structure consists of data variables and one address pointer say next, pointing to the next node.
  + The last node has the next pointing to the first node
  + It is accessed by a pointer, say start, pointing to any node

**Traversing circular linked list**

* The advantage of coircular linked liusts is that it is efficient to traverse the circualar linked list cyclically. Now we look into algorithms of some operations on circular linked lists.

Algorithm: traverse a circular linked list m times, time: O(mn), space: O(1)

Input: start, m

1. Ptr = start, k=0
2. If ptr==NULL goto step 7
3. If k==m goto step 7
4. Process data ptr->data
5. If ptr->next == start, k=k+1, goto step 3
6. Ptr = ptr->next, goto step 4
7. Stop

**Circular doubly linked lists**

* Properties:
  + The next of the last node points to the first node
  + The prev of the first node points to the last node
  + It has one access pointer, say start, pointing to any node