**Unit: 5 Linked List**

1. **Concept and Definition**

**What are the main drawbacks of using sequential storage to represent stacks and queues?**

* Array is implemented using static method. It should be declared before using it. If an array consist only two or three element, other remaining space will be waste.
* Size of array is always fixed. If the requirement of array is increased, we can not extend the size of array while it is using.

Another way to organize a collection of elements is a linked list, or in short list. A list is a collection of nodes; Each node stores an element, and a link to another node. Each item in the list is called **node** and contains two field, an **information** field and a next **address** field. The information field holds the actual element on the list. 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, known as a pointer.The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.



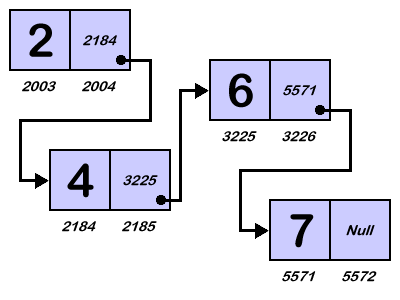


Fig 5.1(b) : Example of Linked List

* The list with no nodes on it is called the empty list or null list.

**Notation for use in Algorithm**

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.

1. **Types of linked list**
   1. **Singly Linked List:**

A linked list is also known as singly link list which is described above.

* 1. **Doubly Linked List**

A doubly linked list is a list that has two references, one to the next node and another to previous node.

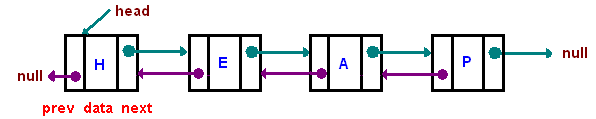


Fig 5.2(c) : Example of Doubly Linked List

* 1. **Circular Linked List**

Another important type of a linked list is called a circular linked list where last node of the list points back to the first node (or the head) of the list.



Fig 5.2(d) : Example of Circular Linked List

1. **Inserting and deleting nodes**

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

**Inserting a nodes**

For example, the list of integer given are shown in figure 5.3.1.a and we are going to add an integer 6 to the front of the list. we change the list so that it will appear as shown in figure 5.3.1 f. The first step is to allocate memory to store 6. Let us assume, this could be achieved by operation

p = getnode();

**Fig : From book(pg188)**

obtains an empty node and set the contents of a variable named p to the address of that node. The value of p is pointer to this newly allocated node. Fig 5.3. 1b illustrate the list and the new node after performing the getnode operation.

The next step is to insert the integer 6 into the info portion of the newly allocated node. This is done by the operation

info(p) = 6;

The result of this operation is illustrated in figure 5.3 .1c

After setting the info portion of the 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, the node that follows should be the current first node on the list. Since the variable list contains the address of that first node, node(p) can be added to the list by performing the operation

next(p) = list;

this operation places the value of list(which is the address of the first node on the list) into the next field of node(p). figure 5.3.1 d illustrate the result of this operation.

At this point p points to the list with the additional item included. However, since list is the external pointer to the desired list, its value must be modified to the address of the new first node of the list. This can be done by performing the operation

list = p;

Which changes the value of list to the value of p. Figure 5.3.1e illustrate the result of this operation. Figure 5.3.1e and f are identical except that the value of p is not shown in Figure 5.3.1 f. This is because p is used as an auxiliary variable during the process of inserting 6 on the list.

Putting all the step together, we have an algorithm for adding the integer 6 to the front of the list list:

p=getnode();

info(p) = 6;

next(p) = list;

list = p;

To insert any element on list , simply replace 6 by x. It also work even list is initially empty ( list == null ).

**Deleting a nodes**

Figure 5.3.2 illustrates the process of removing the first node of a nonempty list and storing the value of its info field into a variable x. The initial and final configuration are shown in fig 5.3.2a and 5.3.2.f The process itself is almost the exact opposite of the process to add a node to the front of the list.

Figure pg 190.

To obtain Figure 5.3.2 d from figure 4.3.3 a, the following operations are performed.

p=list; ( **Figure 5.3.2 b )**

list = next(p); ( **Figure 5.3.2 c )**

x=info(p); ( **Figure 5.3.2 d )**

Now p is not accessible from anywhere in the list. freenode(p) (figure 5.3.e) will deallocate the space allocated by node p and can be reallocated by using operation freenode(p).

Another way of thinking of getnode and freenode is that getnode creates a new node , whereas freenode destroys a node.

1. **Linked implementation of a stack (PUSH / POP)**

Linked implementation of stack is achieved by adding an element from the front of the linked list which is similar to the PUSH operation of stack. A stack is only accessible from the top, and a list can be accessed only from the pointer to its first element. The operation of removing the first element from the linked list is analogous to popping a stack. In both cases the only immediately accessible item of a collection is removed, and the next item becomes immediately accessible.

**PUSH Operation:**

In this way, stack can be implemented in another way. In a linear linked list implementation of stack,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, elt) may be implemented by

p = getnode();

info( p) = elt;

next( p ) = s;

s = p;

**POP Operation:**

The operation empty is just only a test of whether s equal null. The operation x= pop(s) remove the first node from a non empty list and display overflow, if the list is empty.

if ( empty(s)) {  
 printf(“Stack Underflow”);

exit(1);

}

else

{

p=s;

s=next(p);

x=info(p);

freenode(p);  
}

Figure 5.4.a illustrate a stack implemented as a linked list, and figure 5.4 b illustrates the same stack after another element has been pushed onto it.

figure from book pg:192

The advantage of list implementation of stacks is that stacks grow and shrink while inserting and deleting value on it. The space is allocated for each node before inserting value and the space will free when it is deleted. No space has been pre allocated to any single stack and no stack is using space that it does not need.

**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 == null)

{

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 \*/

1. **Linked implementation of a queue (Insert / Remove)**

We know that items are deleted from the front of a queue and inserted at the rear. Let a pointer to the first element of a list represent the front of the queue q.front. Another pointer to the last element of the list represents the rear of the queue q.rear. empty(q) and x=remove(q) are similar to empty(s) and x=pop(s) of stack respectively. Both q.front and q.rear must be null in an empty queue. Disadvantages - A node in a linked list occupies more storage than a corresponding element in an array. The additional time spent in managing the available list for each addition and deletion of an element. All the stacks and queues of a program have access to the same free list of nodes. Nodes not used by one stack may be used by another. An item is accessed in a linked list by traversing the list from its beginning. To access nth item, list implementation requires an operations. It is necessary to pass through each of the first (n-1) elements before reaching the nth item.Advantages - 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.

**The pseudo code for deletion is below:**

*if (empty(q))*

*{*

*printf("Queue is Underflow");*

*exit(1);*

*}*

*p = q.front;*

*t = info(p);*

*q.front = next(f);*

*if (q.front == null)*

*q.rear = null;*

*freenode(f);*

*return(t);*

**The operation insert algorithm is implemented**

*p = getnode();*

*info(p) = x;*

*next(p) = null;*

*if (q.rear == null)*

*q.front =p;*

*else*

*next(q.rear) = p;*

*q.rear = p;*

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 descending 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, whereas this is impossible for an array unless extra space is left empty.

1. **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.

* 1. Stack as a circular list (PUSH / POP)

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\*/*

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 \*/*

1. Queue as a circular list (Insert / Remove)

|  |  |  |
| --- | --- | --- |
| 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.  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\*/*  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  **Primitive operation Of Circular List**  Abstract Data Type (ADT) is a useful tool for specifying the logical properties of data type. An ADT is a collection of values and a set of operations on those values. Mathematically speaking, “a TYPE is a set,, and elements of set are Values of that type”.  **ADT List**  A list of elements of type T is a finite sequence of elements of type T together with the operations of create, update, delete, testing for empty, testing for full, finding the size, traversing the elements. In defining Abstract Data Type, we are not concerned with space or time efficiency as well as about implementation details. The elements of a list may be integers, characters, real numbers and combination of multiple data types.  **Primitive Operation of Singly List** |  |  |

1. Doubly Linked List (Insert / Remove)

In a singly linked list, each element contains a pointer to the next element. We have seen this before. In single linked list, traversing is possible only in one direction. Sometimes, we have to traverse the list in both directions to improve performance of algorithms. To enable this, we require links in both the directions, that is, the element should have pointers to the right element as well as to its left element. This type of list is called doubly linked list.

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

1. **Advantages of linked List:**
   1. Linked lists are a dynamic data structure, allocating the needed memory when the program is initiated.
   2. Insertion and deletion node operations are easily implemented in a linked list.
   3. Linear data structures such as stacks and queues are easily executed with a linked list.
   4. They can reduce access time and may expand in real time without memory overhead.
2. **Disadvantages of linked List:**
   1. They have a tendency to waste memory due to [pointers](http://en.wikipedia.org/wiki/Pointer_(computer_science)) requiring extra storage space.
   2. Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
   3. Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list.
   4. Difficulties arise in linked lists when it comes to reverse traversing. Singly linked lists are extremely difficult to navigate backwards, and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.
3. A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.
4. One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.
5. Another disadvantage is that a linked list uses more memory compare with an array - we use extra 4 bytes (on 32-bit CPU) to store a reference to the next node.
6. Linked lists have the following (dis)advantages when compared with sequential allocation for
7. storing linear lists:
8. 1. Easy insertion of a new node anywhere in the list.
9. 2. Easy deletion of any node of the list.
10. 3. Slow access to random nodes.
11. 4. Extra storage required for links (but often more efficient overall use of store).
12. 5. In general, slightly less efficient scanning through the list (dependent on the instruction sets
13. of particular machines).
14. 6. Two lists may be easily concatenated into a single list, or a single list split into two lists.
15. 7. There is no pre-specified maximum list length.
16. It may be noted also that linked lists can represent structures more complex than linear lists.

Problems:

* What are the differences between a linked list and an array?
* Write a small program to delete the last node of a given linked list.
* Using the same code, convert the existing list into a circular linked list.
* Using the data given in question 2,Write a small function to traverse the linkedlist only once and find out the middle element.
* Given a singly linked list, write a small subroutine getNth() to get nth node of the linked list.
* Write a small subroutine insertNth() to insert nth node in a singly linked list.
* What is a queue and how can you implement queue using a linked list?
* What is a stack and how can you implement stack using a linked list?
* 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.
* Given two singly linked lists, how do you append them?