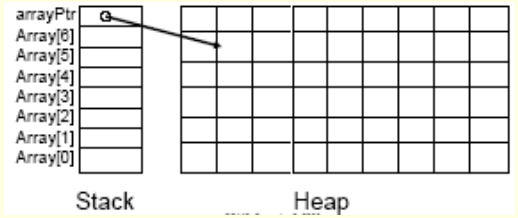
Lecture 9.1

Linked List

What we know about arrays

- Arrays need a continuous block of memory
- int Array[7]={0};
- int *arrayPtr = new int[7];



Considerations

- What happens if we needed a smaller array than the one declared?
- What happens if we needed a bigger array than the one declared?
- What if there is not a continuous block of memory big enough to allocate the array?
- We need a dynamic structure!

Dynamic data structures

- This kind of structures can grow and shrink during execution.
- Linked Lists
- Stacks
- Queues
- Trees

What are linked lists?

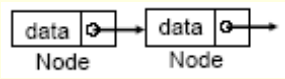
- Way to store data with structures so that the user can automatically create a new place to store data whenever necessary.
- The programmer writes a struct or class definition that contains variables holding information about something, and then has a pointer to a struct of its type.

What are linked lists? (cont)

- Each of these individual struct or classes in the list is known as a node and each node contains whatever data you are storing along with a pointer (a link) to another node

Example:

```
struct Node{
   int data;
   Node * next;
};
```

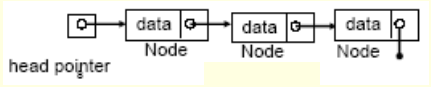


How Linked Lists work?

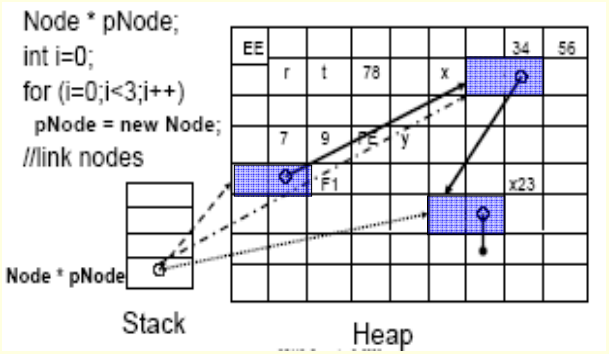
- Think of it like a train.
- The programmer always stores the first node of the list. This would be the engine of the train (head).
- The pointer is the connector between cars of the train.
- Every time the train adds a car, it uses the connectors to add a new car.
- The programmer uses the keyword new to create a pointer to a new struct or class.

Traversing the list

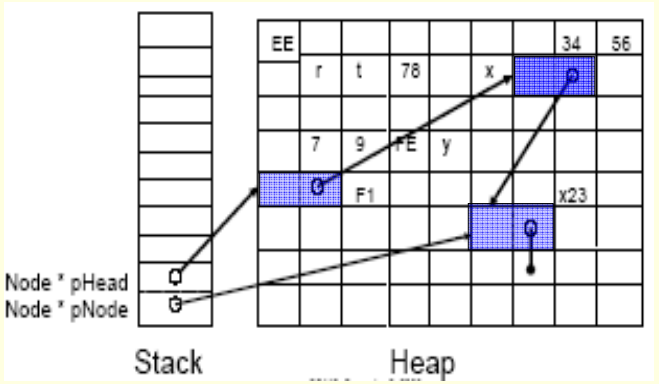
- By locating the node referenced by the head pointer and then doing the same with the pointer in that new node and so on, you can traverse the entire list.
- At the end, there is nothing for the pointer to point to, so it does not point to anything. It should be set to "NULL" to prevent it from accidentally pointing to a totally arbitrary and random location in memory (which is very bad).



How's the memory for a linked list allocated?



Keeping track of the head



Linked Lists: Some Properties

- Linked list basic operations:
  - Search the list to determine whether a particular item is in the list
  - Insert an item in the list
  - Delete an item from the list

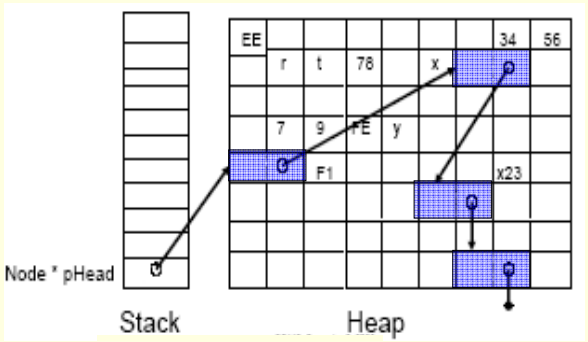
Inserting Items into the list

- New nodes can be inserted into the list at any place:
  - before first node
  - after the final node
  - between two existing nodes
  - into an empty list

Removing items from the list

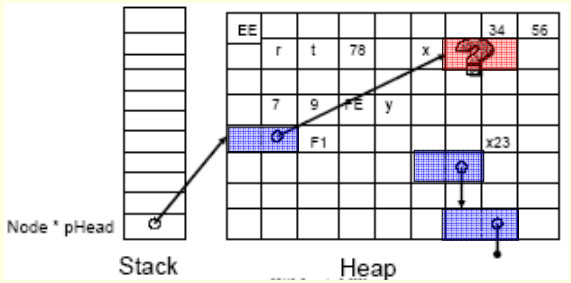
- Nodes may be removed from any place along the chain
  - the first node in the chain
  - final node in the chain
  - between two existing nodes

List before remove



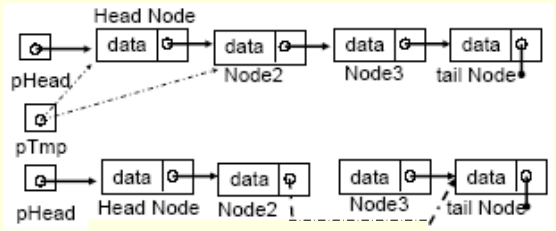
Removing example

- Can you remove a node without any consequences?



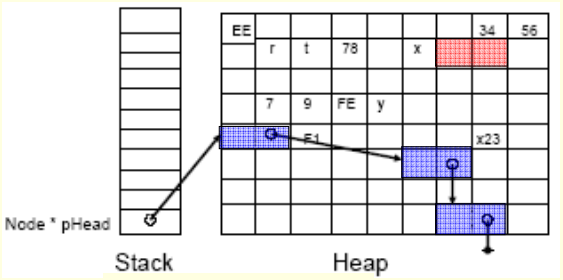
Removing example:

- To remove Node3 we need to know what node is before Node3 to change the “next” pointer.



Removing example

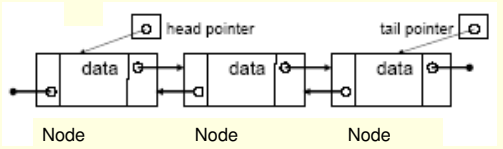
- Keeping track of the previous node would make it easy to find the next node.



Doubly linked list

- Makes it easy to keep track of **previous** and **next** nodes.

```
struct Node {
    int Data;
    Node *prev;
    Node *next;
};
```

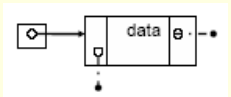


Doubly linked lists (cont)

- These lists still need to code for special cases of inserting at the front, in the middle or after the last node in the chain.
- Similar consideration is required for removing a node from the list.

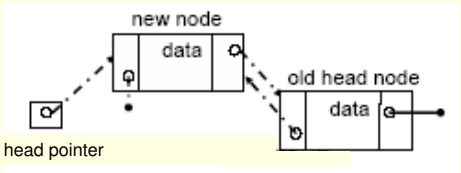
Inserting into an empty list.

1. Create new node
2. Make head pointer point to new node
3. Make “prev” of new node point to null.
4. Make “next” of new node point to null.



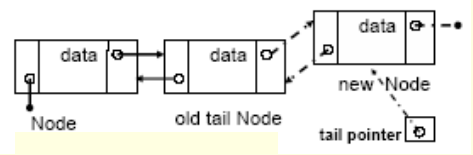
Insert before the first node

1. Create new node
2. Make head node “prev” point to new node
3. Make new node “next” point to head node
4. Make head pointer point to new node
5. Make new node “prev” point to null



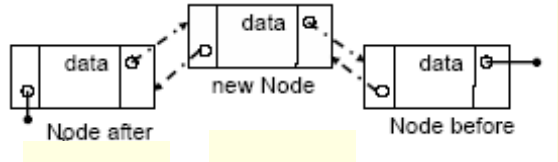
Append at the end of the list

1. Create new node
2. Make tail node “next” point to new node
3. Make new node “prev” point to tail node
4. Make tail pointer point to new node
5. Make new node ”next” point to null.



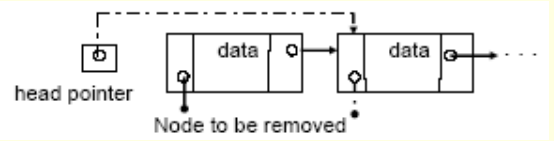
Insert between two existing nodes

1. Create new node
2. Make new node “next” point to “next” of “node after”
3. Make new node “prev” point to “node after”
4. Make “prev” of “node before” point to new node
5. Make “next” of “node after” point to new node



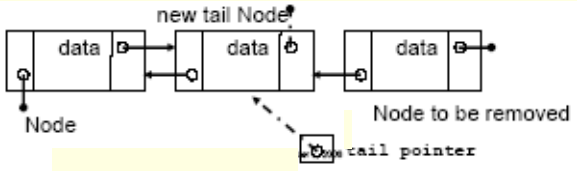
Removing the first node in the list

1. Make head pointer point to “next” of node to be removed
2. Make new head node “prev” point to null
3. Remove the node



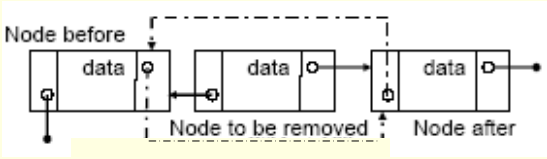
Removing the last node in the list

- Make tail pointer point to the “prev” of node to be removed
- Make “next” of new tail node point to null.
- Remove the node



Removing a node between two existing nodes

- Make “next” of “node before” point to “next” of node to be removed
- Make “prev” of “node after” point to “prev” of node to be removed
- Remove the node



Advantages of a Linked List

- You can quickly insert and delete items in a linked list.
- Inserting and deleting items in an array requires you to either make room for new items or fill the "hole“ left by deleting an item.
- With a linked list, you simply rearrange those pointers that are affected by the change.

Disadvantages of a Linked List

- They are quite difficult to sort.
- You cannot immediately locate, say, the hundredth element in a linked list the way you can in an array. Instead, you must traverse the list until you've found the hundredth element.