Lecture 4: Lists

Data Structure and Algorithm Analysis

Review on Structures

- Structures are aggregate data types built using elements of primitive data types.
- Structure is defined using the struct keyword:

E.g.
```
Struct Time {
    int hour;
    int minute;
    int second;
    };
```

- The struct keyword creates a new user defined data type that is used to declare variables of an aggregate data type.
- Structure variables are declared like variables of other types.
  - Syntax: struct<structure tag> <variable name>;
  - E.g.
    ```
    Struct Time timeObject,
    Struct Time *timeptr
    ```

Accessing Members of Structure Variables

- The Dot operator (.):to access data members of structure variables.
- The Arrow operator (->): to access data members of pointer variables pointing to the structure.
- E.g. Print member hour of time Object and timeptr.
  - cout<< timeObject. hour; or
  - cout<< timeptr->hour;

- Note :timeptr->hour is the same as (*timeptr) . hour
- The parentheses is required since (*) has lower precedence than (.)

Self-Referential Structures

- Structures can hold pointers to instances of themselves.

```
Struct list {
char name[10];
Int count;
Struct list *next;
};
```

- However, structures cannot contain instances of themselves.

The List ADT

- A list data structure is sequence of zero or more elements

$$A_1, A_2, A_3, \ldots A_N$$

  - N: length of the list
  - A_1: first element
  - A_N: last element
  - A_i: element at position i
  - If N=0, then empty list
  - Linearly ordered
    - A_i precedes A_{i+1}
    - A_i follows A_{i-1}

Common operations of the List data structures

- printList: print the list
- makeEmpty: create an empty list
- find: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) → 3
- insert: insert an object to a list
  - insert(x,3) → 34, 12, 52, x, 16, 12
- remove: delete an element from the list
  - remove(52) → 34, 12, x, 16, 12
- findKth: retrieve the element at a certain position

Implementation of an ADT

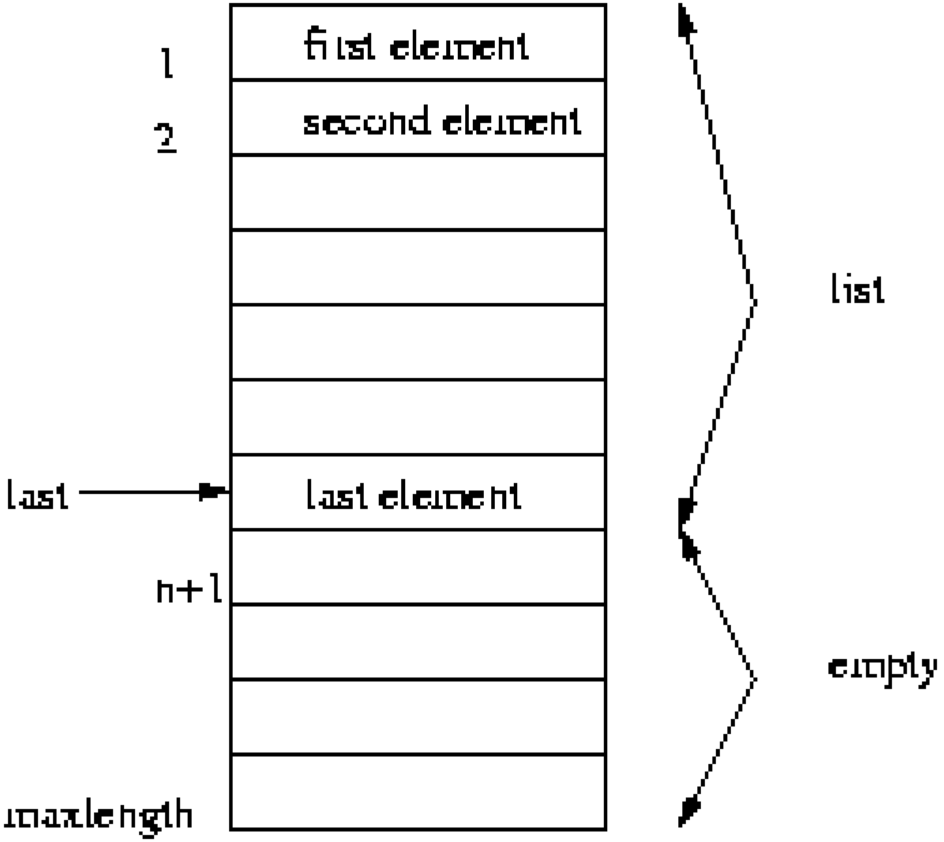
- Choose a **data structure** to represent the list ADT
  - E.g. arrays, LinkedList etc.
- Each operation associated with the ADT is implemented by one or more subroutines(functions)
- Two standard implementations for the list ADT
  - Array-based
  - Linked list

Array Implementation

- Need to know the maximum number of elements in the list at the start of the program
  - Difficult
  - Wastes space if the guess is bad
- Adding/Deleting an element can take O(n) operations if the list has n elements.
  - As it requires shifting of elements
- Accessing/changing an element anywhere takes O(1) operations independent of n
  - Random access

Array Implementation

- Elements are stored in contiguous array positions



Adding an element

- Normally first position (A[0]) stores the current size of the list
- Actual number of elements currsize+ 1
- Adding at the beginning:
  - Move all elements one position up/behind
  - Add at position 1;
  - Increment the current size by 1

```
For (j = A[0]+1; j > 1; j--)
    A[j] = A[j-1];
A[1] = new element;
A[0]=A[0]+1;
```

Complexity: O(n)

Adding at the End

- Add the element at the end
- Increment current size by 1;

```
A[A[0]+1] = new element;
A[0]=A[0]+1;
```

- Complexity: O(1)

Adding at kth position

- Basic Steps
  - Move all elements one position behind, kth position onwards;
  - Add the element at the kth position
  - Increment current size by 1;

```
For (j = A[0]+1; j > k; j--)
    A[j] = A[j-1];
A[k] = new element;
A[0]=A[0]+1;
```

- Complexity: $O(n-k)$

Deleting an Element at the Beginning

- Deleting at the beginning:
  - Move all elements one position ahead;
  - Decrement the current size by 1

```
For (j = 1; j < A[0] ; j++)
    A[j] = A[j+1];
A[0]=A[0]-1;
```

- Complexity: O(n)

Deleting at the End

- Delete the element at the end
- Decrement current size by 1;

A[0]=A[0]-1;

- Complexity: O(1)

Deleting at the k^{th} position

- Basic Steps
  - Move all elements down one position ahead, k+1th position onwards;
  - Decrement the current size by 1;

```
For (j = k; j < A[0]+1; j++)
        A[j] = A[j+1];
A[0]=A[0]-1;
```

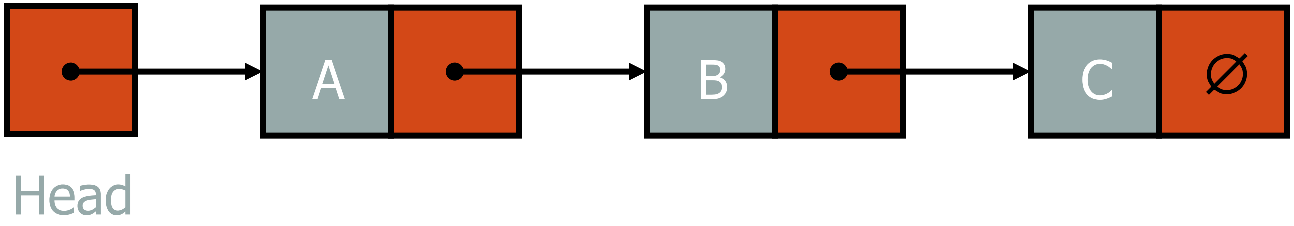
- Complexity: $O(n-k)$

Accessing an Element at the kth position

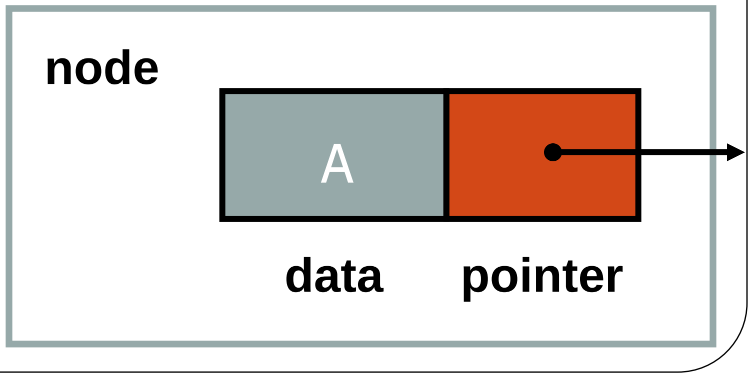
A[k];

- O(1) operation;

Linked Lists implementation



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`



Array Vs. Linked list

Array	Linked Lists
•Physically Contiguous	•Logically Contiguous Only
•Fixed Length	•Changeable Length
•Access Elements by Index	•Access Elements by Traversal
•Insertion/Removal is Costly	•Insertion/Removal is Efficient

Defining the data structure for a linked list

- The key part of a linked list is a structure, which holds the data for each node. Example,
  - name, address, age or whatever for the items in the list and,
  - most importantly, a pointer to the next node.

- Example of a typical node:

```
Struct node{
    char name[20]; // Name of up to 20 letters
    Int age;
    float height; // In metres
    node *next; // Pointer to next node
};
Struct node *head= NULL;
```

Operations on Linked lists

- **Inserting a node**
  - **At the beginning**
  - **At the end**
  - **At kth position**
- **Removing Elements**
  - **From front**
  - **From end**
  - **From kth position**
- **Traversing the list**

Adding an element at the beginning

- Create a new node;

```
Struct node *newnode
newnode= new node;
```

- Fill in the details

```
newnode-> name = // store the value of the name field
newnode-> age=  // store the value of the age field
newnode-> height= // store the value of the heightfield
newnode->next = NULL
```

- if the list is empty to start with,

```
 if (head== NULL) head = newnode;
```

- Else Pointer from the newnode points to head;

```
newnode->next= header;
header= newnode;
```

Adding an element at the end

- Create a new node;
- Pointer from the last node points to new node;

```
Create(newnode);
last.next= newnode;
```

- How do we find the last node?
- Soln: step through the list until it finds the last node.

```
last = head; // We know this is not NULL -list not empty!
while (last->nxt!= NULL){
last= last>nxt; // Move to next link in chain
}
```

Adding an element at the end…

- Adding an element at the end

```
void add_node_at_end() {
node *temp, *temp2; // Temporary pointers
// Reserve space for new node and fill it with data
temp = new node;
cout<< "Please enter the name of the person: ";
cin>> temp->name;
cout<< "Please enter the age of the person : ";
cin>> temp->age;
cout<< "Please enter the height of the person : ";
cin>> temp->height;
temp->nxt= NULL;
// Set up link to this node
if (head == NULL)
head = temp;
else {
temp2 = head ;
// We know this is not NULL -list not empty!
while (temp2->nxt!= NULL)
{
temp2 = temp2->nxt;
// Move to next link in chain
}
temp2->nxt= temp;
}
}// add node at end
```

Complexity: O(n)

Displaying the list of nodes

- Method

1. Set a temporary pointer to point to the head
2. If the pointer points to NULL, display the message "End of list“ and stop.
3. Otherwise, display the details of the node pointed to by the head pointer.
4. Make the temporary pointer point to the same thing as the nxt pointer of the node it is currently indicating.
5. Jump back to step 2.

Displaying the list of nodes

```
temp = head;
do {
if (temp = = NULL)
cout<< "End of list" << endl;
else
{// Display details for what temp points to
cout<< "Name : " << temp->name << endl;
cout<< "Age : " << temp->age << endl;
cout<< "Height : " << temp->height << endl;
cout<< endl; // Blank line
// Move to next node (if present)
temp = temp-> nxt;
}
} while (temp != NULL);
```

Navigating through the list

- Necessary when you want to insert or delete a node from somewhere inside the list

```
node *current;
current = head;
if (current -> nxt= = NULL)
cout<< "You are at the end of the list." << endl;
else
current = current-> nxt;
```

- Moving the current pointer back one step is a little harder

Deleting a node

- Basic steps
  - Find the desirable node (node to be deleted)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node
- When it comes to delete nodes, we have three choices:
  - Delete a node from the start of the list,
  - Delete one from the end of the list, or
  - Delete at the kth position

Deleting the first node in the linked list

- temp = head; //Make the temp pointer //point to the head pointer
- head = head->nxt; // Second node in chain
- Delete temp;

- Here is the function that deletes a node from the head:

```
void delete_start_node()
{
    node *temp;
    temp = head;
    head= head-> nxt;
    delete temp;
}
```

Deleting the last node

- Steps:

1. Look at the head pointer.
    a. If it is NULL, then the list is empty, so print out a "No nodes to delete" message.

2. Make temp1point to whatever the head pointer is pointing to.

```
temp1=head;
```

3. If the nxtpointer of what temp1indicates is NULL, then we've found the last node of the list, so jump to step 7 otherwise go to the next step.

```
if(temp1->next==NULL)
```

4. Make another pointer, temp2, point to the current node in the list.

```
temp2=temp1
```

5. Make temp1point to the next item in the list.

```
temp1=temp1->next
```

6. Go to step 3.

7. Delete the node pointed by temp1.

```
delete temp1
```

8. Mark the nxtpointer of the node pointed by temp2 as NULL -it is the new last node.

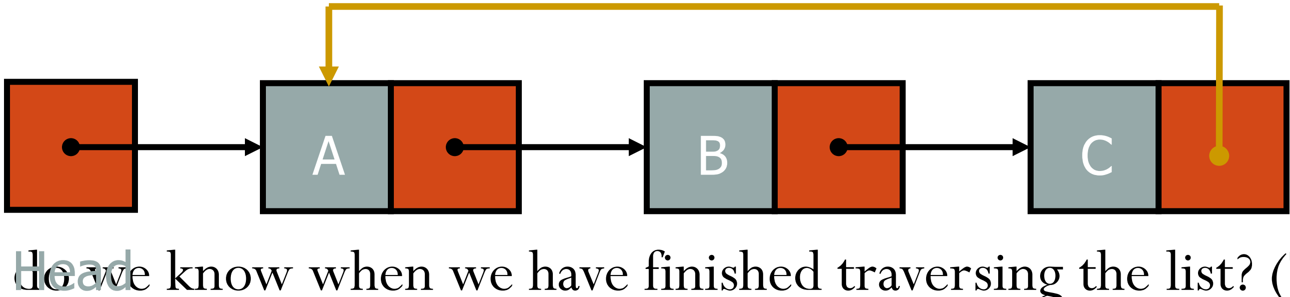
```
temp2->next=NULL
```

Delete node from the end of the list

```
void delete_end_node()
{
node *temp1, *temp2;
if (head == NULL)
cout<< "The list is empty!" << endl;
else {
temp1 = head;
if (temp1->nxt== NULL)
{
delete temp1;
head = NULL;
}
else {
while (temp1->nxt!= NULL) {
temp2 = temp1;
temp1 = temp1->nxt;
}
delete temp1;
temp2->nxt= NULL;
}
}
} // delete end of node
```

Variations of Linked Lists

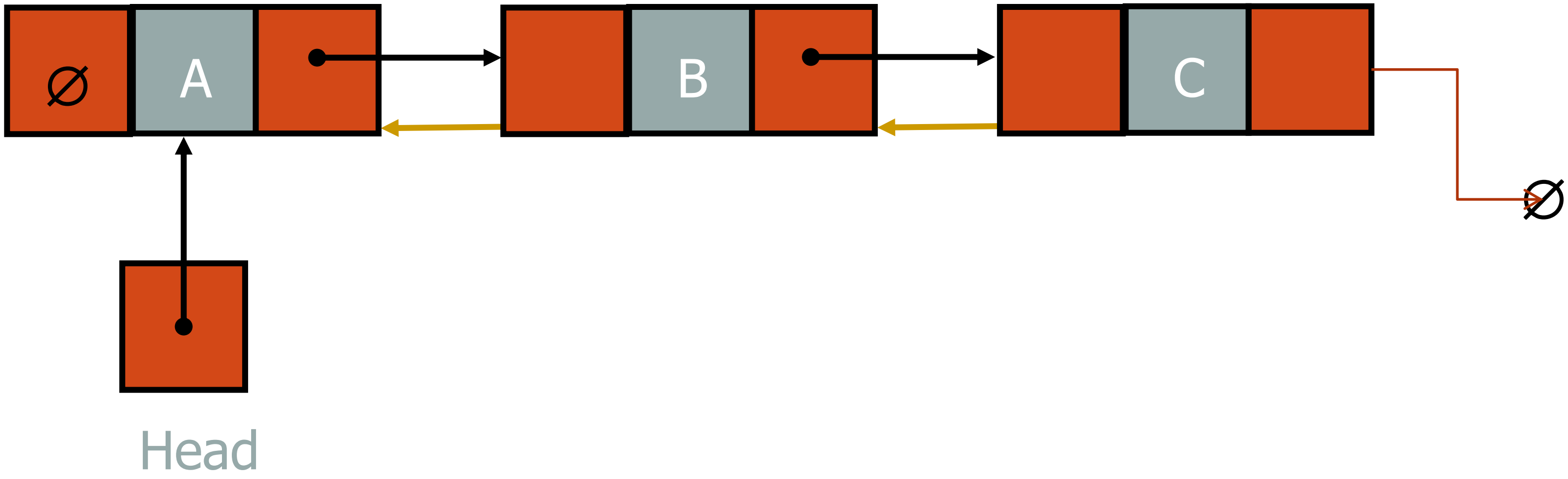
- *Circular linked lists*
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

Variations of Linked Lists

- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL`: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Creating Doubly Linked Lists

- The nodes for a doubly linked list would be defined as follows:

```
struct Node
    {
    int data;
    struct Node *Next;
    struct Node *Prev;
    }*Head;
```

- Data a new node can be created as follows

```
Node *current;
current = new node;
current->data= 15;
current->nxt= NULL;
current->prv= NULL
```

- Finally, link the node in the list

Add node at the beginning of the list

```
void Insert_front(int num)
   {
     struct Node *temp;
      temp = new Node;
     temp->Data = num;
     if (Head == NULL)
     {
        //List is Empty
        Head=temp;
        Head->Next=NULL;
        Head->Prev = NULL;
     }
     else
     {
        temp->Next=Head;
        Head->Prev = temp;
        Head=temp;
     }
   }
```

Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.

Exercise

- Write full implementation for doubly linked lists and Circular lists.
- Your implementation should support the following operations
  - Adding element/node
    - At the beginning
    - At the end
    - At the middle/specific location
  - Deleting data/node
    - From front
    - From end
    - From middle
  - Displaying the list elements

End of lecture 4

The Next Lecture:-Stack and Queue