Linked List[1]

R. Inkulu
http://www.iitg.ac.in/rinkulu/

[1] C code is developed on the board (and the same is not available in slides)

Introduction

Motivation for insertion/deletion within a sequence:

- sorted keywords together with additions
- a polynomial representation that requires updates

Linked list compises of nodes; each node has zero or more primitive/custom data objects (or, pointers) together with one or more links to other nodes. Essential operations to be supported include -

- *inserting a node* - at the beginning, at the end, in-between
- *deleting a node* - at the beginning, at the end, in-between
- *locate data* - based on key comparison, locate the node referring to the said data
- *traverse* the list - typically to apply a function over (satellite) data

Singly-linked list (a.k.a. chain)

Disadvanatages of a (dynamic) array:

- contiguous memory requirement
- resizing involves bulk copy
- deleting/inserting an element r in the middle is inefficient: may involve cleaning up to bring valid elements together

Motivation for singly-linked list:

- no contiguous memory requirement for any two successive nodes
- a node is allocated whenever it need to be introduced
- when a node r need to be deleted, only traversal to r is needed; does not involve moving of entry contents

Disadvantages of a singly-linked list:

- traversal is sequential, and hence slower
- cannot exploit locality based caching
- maintaining and storing pointers with each block

(Singly-)linked list

```
struct NodeA {
   void *data;
   struct NodeA *next; //self-referential
};
typedef struct NodeA Node;
typedef void (*TraverseHelper)(void*);
typedef int (*LocateHelper)(void*, void*);

Node *insertNodeAtEnd(
   Node *head, void *p);
Node *insertNode(Node *head,
   void *p, int location);
int locateData(Node *head, void *p, LocateHelper func);
Node *deleteNode(Node *head,
   void *p, LocateHelper func);
void traverseList(Node *head, TraverseHelper func);
void deleteList(Node *head);
```

(Singly-)linked list vs Doubly-linked list

Disadvantages of a singly-linked list:

- traversing nodes in reverse order cannot be done without using a stack
- given a pointer p to a node, it is not possible to traverse nodes preceding p in the list

Disadvantages of a doubly-linked list:

- additional pointer to previous node need to be maintained and saved with each node

Doubly-linked list

```
struct NodeA{
   void *data;
   struct NodeA *next;  //self-referential
   struct NodeA *prev;  //self-referential
};
typedef struct NodeA Node;
typedef void (*TraverseHelper)(void*);
typedef int (*LocateHelper)(void*, void*);
```

```
Node *insertNodeAtEnd(
   Node *head, void *p);
Node *insertNode(Node *head,
   void *p, int location);
int locateData(Node *head, void *p, LocateHelper func);
Node *deleteNode(Node *head,
     void *p, LocateHelper func);
void traverseList(Node *head, TraverseHelper func);
void traverseListInReverse(Node *head,
     TraverseHelper func);
void deleteList(Node *head);
```

Worst-case time complexity of doubly-linked list functions

- insertNode $O(n)$
- deleteNode $O(n)$
- traverse $O(n)$ (excluding the execution func over data)
- delete $O(n)$

homework: analyze the tight worst-case asymptoic upper bound on the space complexity homework:

- Implement dynamic-sized stack to store pointers to satellite data using a doubly-linked list and analyze the worst-case time complexity of push and pop.

- Implement dynamic-sized deque to store pointers to satellite data using a doubly-linked list and analyze the worst-case time complexity of appropriate operations.

Circular linked list

Motivation: implementing a round robin algorithm

Homework: Devise a ADT for circular linked list and analyze the asymptotic worst-case time and space complexities of its operations